UNIFICATION OF LARCH AND Z-BASED
OBJECT MODELS
TO SUPPORT ALGEBRAICALLY-BASED
DESIGN REFINEMENT:
THE Z PERSPECTIVE

THESIS

Kathleen May Wabiszewski
Captain, USAF

AFIT/GCS/ENG/94D-24

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-24

UNIFICATION OF LARCH AND Z-BASED OBJECT MODELS TO SUPPORT ALGEBRAICALLY-BASED DESIGN REFINEMENT: THE Z PERSPECTIVE

THESIS

Kathleen May Wabiszewski
Captain, USAF

AFIT/GCS/ENG/94D-24

Approved for public release; distribution unlimited

AFIT/GCS/ENG/94D-24

Unification of Larch and Z-Based Object Models To Support Algebraically-Based Design Refinement: The Z Perspective

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science

Kathleen May Wabiszewski, B.S., M.S.

Captain, USAF

December 13, 1994

Approved for public release; distribution unlimited

Acknowledgements

I would like to thank my advisor, Major Paul Bailor, for his guidance and assistance during this research effort. I also wish to thank my committee members, Dr. Gary Lamont and Major David Luginbuhl for their help and advice.

I also wish to thank the other members of the Knowledge Based Software Engineering (KBSE) research group who offered words of encouragement and humorous, i.e. "geeky", diversions when I needed them most. Thanks to my research partner, Captain Cathy Lin, for her contributions to the successful results that we achieved.

I would also like to thank some members of the European *Z* community that provided their assistance and expertise in this undertaking: Dr. John McDermid and Dr. Ian Toyn of York Software Engineering, and Dr. Samuel Valentine of the University of Brighton. Through email messages and a brief meeting at the *Z* User Meeting in Cambridge, I was able to gain valuable knowledge about *Z* that bolstered the acceptability of my work.

Finally, I would like to thank my husband, Mike, and my daughter, Caitlin, for all of their support and understanding. In a little over three years, I have dedicated a lot of time to my education and, subsequently, completed two Master's degrees. I appreciate my family's sacrifices and am looking forward to more time together in the very near future.

Kathleen May Wabiszewski

Table of Contents

Page

List of Figures

Figure		Page

Figure | Page

Figure	Page

List of Tables

AFIT/GCS/ENG/94D-24

Abstract

This research established a foundation for formalizing the evolution of Z-based object models to theories, part of a dual approach for formally extending object-oriented analysis models using the Z and LARCH languages. For the initial phase, a comprehensive, consistent, and correct Z language parser was implemented within the SOFTWARE REFINERY^{TM} Programming Environment. The Z parser produced abstract syntax trees (ASTs) of objects, thereby forming the basis for analyzing the similarities and differences between the Z-based and LARCH-based object representations. The second phase used the analysis of the two languages to identify fundamental core constructs that consisted of similar syntactic and semantic notions of signatures and axioms for describing a problem domain, thereby forming a canonical framework for formal object representations. This canonical framework provides a front-end for producing design refinement artifacts such as synthesis diagrams, theorem proving sentences, and interface languages. The final phase of the process demonstrated the feasibility of interface language generation by establishing an executable framework that mapped Z into the SOFTWARE REFINERY^{TM} Environment to rapidly prototype object-oriented Z specifications.

Unification of Larch and Z-Based Object Models

To Support Algebraically-Based

Design Refinement:

The Z Perspective

I. Introduction

1.1 Background

The evolution of technology has imposed a great challenge upon the software development process: how to accurately and adequately specify a system's desired behavior. The goal of these specifications is the abstract description, independent of implementation details, of the system's functional behavior (41:1). Unfortunately, many software development organizations are still relying on informal, error-prone techniques and the resulting products do not satisfy the customer's requirements. Therefore, in an attempt to change the existing mindset and produce unambiguous, verifiable system specifications, the software engineering community is researching the area of formal methods. These methods, established on provable mathematical constructs, are powerful mechanisms capable of abstractly describing behavior. Based on set theory and predicate calculus, formal methods empower software engineers with the same descriptive and analytical techniques used by traditional engineers while aiding in the evolution of software engineering into a true engineering discipline. Additionally, some formal-based specifications can be executed and/or

transformed to produce viable software components, which in turn can be composed to create the desired system.

Since the realm of formal methods is essentially a branch of applied mathematics, various notations or *languages* have been developed to support their application. The role of the notation is to connect language expressions to a sound foundation in logic (10:38), while exploiting the descriptive properties of the underlying mathematical system. These formal specification languages describe the functionality of a software system in terms of a particular state space, together with a collection of operations that act upon the space (31:5). The operations can be constrained to a range of valid states through preconditions and postconditions, thereby stating "what" the system does and not "how" it performs the task (procedural abstraction) (12:4). Various strategies of coupling a representative notation with fundamental mathematical objects have produced two major perspectives of formal specification generation: theory-based and object-based.

In a theory-based approach, the target system is described using algebraic structures to model its basic operators and the equational properties of its behavior. The equational identities or axioms demonstrate how different combinations of the basic operators can create a collection of structures that satisfy the desired model (31:4). Unfortunately, the mathematically intense nature of this approach has discouraged many software engineers, therefore limiting its widespread use. In parallel with these developments, there has been a less rigorous extension centered around an object-based perspective. In this approach, the target system is modeled as a collection of fundamental objects characterized by their intrinsic properties. This method is well suited for handling the structure of a specification but often introduces ambiguity based on individual interpretations of an object's behavior.

Regardless of this shortcoming, the informality of this process makes it user-friendly and more easily accepted.

In an effort to benefit from both aspects, the research community is investigating various techniques that attempt to meld these two perspectives. For example, one proposed methodology is a domain-oriented application composition system being developed by the Knowledge-Based Software Engineering (KBSE) research group at the Air Force Institute of Technology (AFIT). The KBSE application composition approach uses object-oriented models developed from a variety of modeling methods, e.g., McCain (25) and Rumbaugh (40), and transforms these models into an object-based model within the Software Refinery design and synthesis environment. Written in REFINE, a wide-spectrum, mathematically-based language, these portable object-based models can be used to produce executable specifications. The KBSE application composition system, known as ARCHITECT, uses these canonical formal specifications to create domain-specific applications. For each composed application, ARCHITECT can execute a prototype, thereby providing a means to demonstrate correct behavior.

Despite this seemingly *tailor-made* composition method, this current tactic does not adequately support verifiability using provable constructs. Another generation of application composers is needed to guarantee the soundness of future domain models. Applying a systematic approach based on formal languages and algebraic specifications could provide that guarantee and serve as a solid foundation for developing the next generation application composition system. To that end, Figure 1.1 illustrates the AFIT KBSE group's design of a potential framework for front-end support of a future composition system.

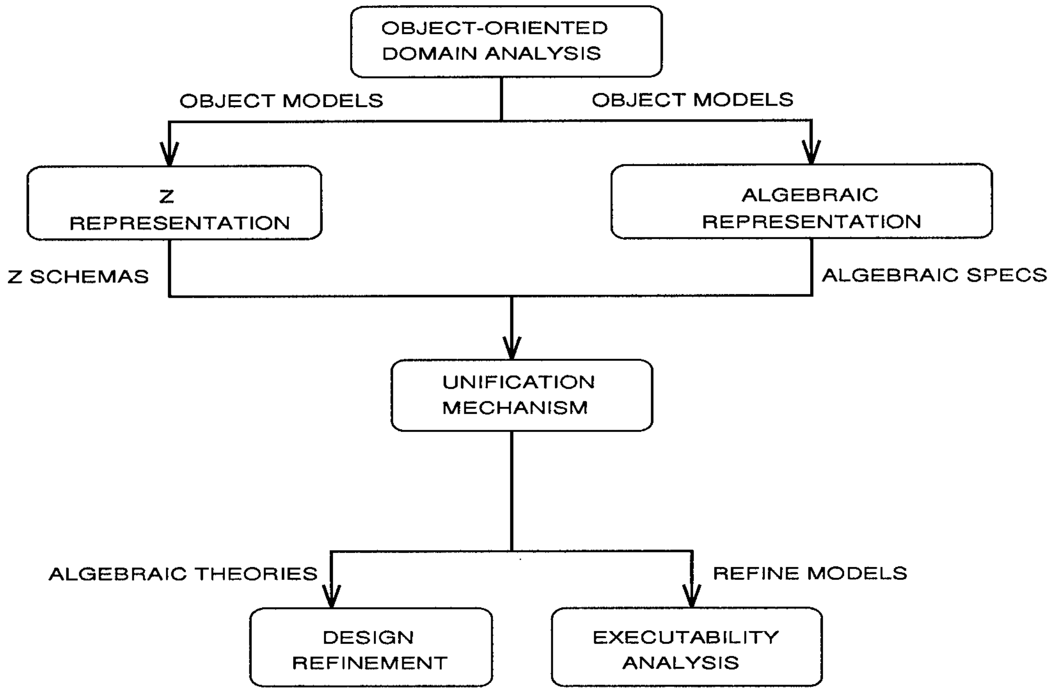

Figure 1.1 Formalized Object Transformations

The proposed framework depicts formalized object transformations through two complementary paths from Object-Oriented Analysis (OOA) into REFINE object base models. The left path is based on the non-executable Z (pronounced "zed") specification language, which is representative of an object/model-based approach. Similarly, the right path is based on algebraic specifications and represents a theory-based perspective. The framework also contains a third *unifying* path between the resulting REFINE object models. Since each side has its foundations in mathematical constructs, these resulting models should theoretically capture the same behaviors for a specified domain. A unifying mechanism permits generalized refinement of both languages within a common abstract framework. Subsequently, this canonical model provides the basis for two target phases: design refinement and executability analysis. The former phase provides a satisfiable interface to

the design process through refinement of abstract specifications into concrete descriptions, while the latter phase permits specification demonstration and validation via an execution framework.

1.2 Problem

The current domain-oriented application composition approach permits the modeling of abstract specification details. Problem complexity is adequately managed through abstraction and encapsulation. However, this methodology does not guarantee correctness throughout the process, since no verification of object interfaces (horizontal and vertical relationships) is currently implemented. By definition, horizontal relationships refer to the interconnection of two units on the same level while vertical relationships are indicated by changing levels through the refinement of units (13:465). Since a major goal of the next generation composition system is the ability to formally map between levels of abstraction, i.e., horizontal and vertical structures, the formalized object transformations of Figure 1.1 represent a viable solution to this problem.

1.2.1 Problem Statement.

> *Formalize the object-based composition approach via the transformation of Z-based domain models into an executable framework, while incorporating a unification with corresponding algebraic-based models.*

1.3 Scope

This research addresses the *Z*-based portion and a merging of the two individual models into a unifying REFINE object base model, while concurrent research by Captain Catherine Lin addresses the algebraic-based path (23). Previous work by AFIT's Hartrum

and Bailor (17) has resulted in a methodology for formally extending OOA models using portions of the Z notation. The resulting specification formalizes the domain model, but the absence of a compiler prevents an automated transformation into the REFINE object base. Therefore, a major objective of this research is the initial development and validation of a comprehensive, consistent, and correct Z language compiler, correlated to the existing OOA framework. A second objective focuses on the development of a formal unifying model based on the structural similarities and differences of the Z and algebraic notations. Using the unified model, the final objective is the design and implementation of an initial execution framework. The design refinement tasks, depicted in the lower left portion of Figure 1.1, will not be addressed during this effort.

1.4 Sequence of Presentation

The remainder of this thesis is organized as follows:

Chapter II provides a review of current literature in the areas of Z language development and tools, object orientation and Z, algebraic specification languages, and implementation of specifications.

Chapter III presents the specific development approach for an object-based Z to REFINE compiler, which incorporates a unification with the designated LARCH algebraic specification language. Additionally, the design validation criteria and sample problem domains are identified.

Chapter IV discusses the design, implementation, and validation of a Z parser, including the Mathematical ToolKit.

Chapter V describes the design, implementation, and validation of a unified domain model and accompanying parsers.

Chapter VI presents the design and development of an initial execution framework for the two source languages, *Z* and LARCH.

Chapter VII contains conclusions about the resultant formalized transformation process as well as recommendations for future research.

Several appendices are included to provide additional information concerning this formalized object transformation process. Appendices A, B, and C contain the selected validation specifications and problem domains. Appendix D depicts the graphical domain model of the unified core, while the accompanying REFINE implementations of the domain model and the State Transition Table are presented in Appendices E and F. Similarly, the graphical models of the *Z*-specific (U*Z*ed) extensions and the representative REFINE code are contained in Appendices G and H, respectively. Appendix I contains the grammars for both the core U*Z*ed language and the Unified version of the Mathematical ToolKit. The code for semantic analysis, i.e., shorthand expansion, is detailed in Appendix J. The graphical domain model of the target REFINE program is contained in Appendix K, while the accompanying code is in Appendix L. Appendix M contains the REFINE implementation of the initial execution framework. Appendix N contains a User's Manual for the U*Z*ed parser and execution framework and a point of contact from whom to acquire the REFINE source code for both the *Z* and U*Z*ed versions.

II. Literature Review

2.1 Introduction

This literature review examines research and information relevant to the design of the *Z*-based object transformation process and unifying model described in Chapter I. Since this thesis merges several separate, but integral software engineering disciplines, the review involved an extensive gathering and evaluation of knowledge for applicability and usefulness. Focusing on the previously established research objectives, the evaluation identified six main topics of interest: *Z* language development, *Z* tools, object orientation and *Z*, *Z* and structured development methods, algebraic specification languages, and the implementation of specifications.

2.2 Z Language Development

Z is a formal language used to specify sequential systems through a model-based approach, whereby a system's states and operations are abstractly represented by mathematical constructs (62:262). Using mathematics to model a system's data permits effective reasoning about its properties and yields provable results or theories about its behavior. The *Z* notation is based on the mathematical disciplines of predicate logic and set theory. The predicate logic provides abstract descriptions of a system's operations and their effects, while set theory produces specifications that are precise, unambiguous, concise, and amenable to proof (12:7). In addition to these disciplines, *Z* employs a *schema calculus* for the systematic encapsulation of a system's static and dynamic aspects. Delimiting these aspects via the schema construct can reduce the complexity of the specification and high-

light any inconsistencies or deficiencies. A system's states and operations are decomposed into static and dynamic schemas, respectively, as follows (45:2):

1. Static schemas
   - The states that a system can occupy.
   - The invariant relationships maintained through state transitions.
2. Dynamic or operation schemas
   - The possible operations in a system.
   - The relationships between operations' inputs and outputs.
   - The state changes that take place.

The model-based approach of the *Z* language is dependent on the construction of complex objects from a collection of fundamental objects and their associated types. This methodology demands that every mathematical expression in a *Z* specification be given a type which determines a set known to contain the value of the expression (45:26). There are essentially four basic objects and types within the *Z* language (31:5):

1. Given sets and basic types - The atomic objects of a specification, without any given internal structure, e.g., integers ($\mathbb{Z}$), yet form the boundary of the specification. These basic types not only permit the construction of the remaining three compound types (see below), they also provide the capability to construct complex, composite objects.

2. Sets and set types - Sets containing objects of a given basic type, e.g., a set of a potentially infinite collection of integers is of type $\mathcal{P}\mathcal{Z}$. Set members may be explicitly or implicitly constructed.

3. Tuples and Cartesian product types - These compound types are generally called $\mathcal{N}$-tuples and may be defined over n sets where n is greater than or equal to 2. Set members may be explicitly or implicitly constructed.

4. Bindings and schema types - Bindings are objects that map variable names to values within a Z specification, and bindings are formally established by schemas (8:190). Bindings are used in the operations of Z and implicitly define instances of schema *types*. The type or *signature* of a schema is a collection of typed variable names. Signatures are created by declarations, and they provide a vocabulary for making mathematical statements, which are expressed by predicates (45:30).

All other mathematical objects used within the Z language are modeled by combinations of these fundamental objects and are formally defined in Spivey's Mathematical ToolKit (45). These prescribed operations on sets and integers are generally regarded as standard in Z, but any specifier is at liberty to either define new sets and operations or rework existing definitions. A notable example is the generalization of numbers in the ToolKit to include the real numbers, $\mathcal{R}$, and the rational numbers, $\mathcal{Q}$, along with the existing integers, $\mathcal{Z}$ (54) (53).

2.2.1 Encapsulation of Theories. As previously stated, the Z language is generally considered to be a model-based approach to software specification. However, since theories model a system's behavior via axiomatic definitions of the operations, the Z language can

also be considered an encapsulation of theories (4). The underlying relationship between the two approaches is analogous to the relationship between an object class (theory-based) and an instance of that object class (model-based). As an instance possesses the same core attributes as the parent object class, similarly, the intrinsic properties of the model-based Z language embody the same mathematical foundations found in theory-based algebraic languages.

2.2.2 "Executability" of Z. The Z specification language is generally non-executable, concentrating on "what" a system does and not "how" it performs the task. The main reason for its lack of executability is caused by the incomplete computation of some of the standard set theory operators within the appropriate abstract domain (8:185). Regardless, executable interpretations of a *subset* of the language are desirable to extend the usefulness of Z as a prototyping language. A prominent example is the interpreter for the Z^{--} language (55) (57) (58) (56). This tool is capable of "executing" a Z specification, where "execution" is defined to be the process of establishing the values of top-level variables that are uniquely determined when the specification is satisfied (58). Z^{--} has its own modified version of the Mathematical ToolKit and attempts to maintain a sound theoretical basis.

Unfortunately, there are many other ad hoc Z utilities available that do not concentrate on the notion of correctness but on other factors listed below (8:186).

- Efficiency - the speed with which a result is obtained.
- Coverage - the portion of the Z grammar that can be successfully handled.
- Sophistication - the termination properties of the technique.

These factors are important to the development of any interpreter or compiler, but correctness should be the primary concern.

2.3 Z Language Tools

As formal methods like *Z* gain acceptance among software developers, the need for language support tools becomes apparent. The precise mathematical syntax of *Z* facilitates the creation of computer-based tool sets for supporting the entire process of formal specification development. This process can benefit most from tools that meet the following criteria (50):

- Support writing specifications
- Support reading specifications
- Prove properties of a specification
- Perform refinement of specifications into code
- Support the Mathematical ToolKit
- Support large specifications

Currently, there are many different types of *Z* language tools available, all of varying quality and robustness. As a representative sample, the following sections summarize the functions and capabilities of four widely used *Z* tools.

2.3.1 CADiZ. CADi*Z* is a UNIX-based suite of tools designed to check and typeset *Z* specifications. It checks for syntactic, scope, and type correctness and then provides diagnostic error reports that are accessible through an interactive browsing com-

ponent (50:11). Through this browser, the user can query a range of semantic properties that includes expansion of schemas, precondition calculations, and simplifications by the one-point rule (26). CADi*Z* also contains a proof editor that utilizes the second edition of Spivey's Mathematical Toolkit in conjunction with the proof rules defined in the *Z* Base Standard (Version 1.0) (9). The proof editor enables the interactive construction of proofs through graphical proof trees. Presently, an experimental extension (called *Z*eta) for refining *Z* specifications to SPARK Ada (a subset of Ada) programs is under development. This module produces the appropriate verification conditions for each refinement step and then interfaces to a theorem proving tool (26).

2.3.2 Formaliser. Formaliser is an interactive, single-user software tool that supports editing, browsing, and type-checking of formal languages. It accommodates various formal languages through supplied attribute grammars that specify the language syntax and its context conditions. In a specific application, Formaliser supports the production of well-formed *Z* specifications via building, editing, checking, and viewing options. Structure editing combined with "on-the-fly" parsing ensures that all Z documents developed within Formaliser are syntactically correct (50:3). Additionally, the tool displays mathematical symbols and constructions as they will appear on the printed page and supports interactive queries about scope and type (14:128). All specification documents are stored within Formaliser's library, which permits the simultaneous editing of multiple documents and textual inclusion for linkage between documents.

2.3.3 Z and HOL. The *Z* language is based on set theory and first order predicate logic and is often used for human-readable formal specifications, while HOL is a

theorem proving environment for higher order logic (7:141). The *Z* and HOL system provide lightweight reasoning support for the *Z* notation through shallow, semantic embedding within HOL. This shallow embedding maps language constructs to their semantic representations in the metalanguage, thus enabling reasoning. This can be contrasted to deep embedding which permits theorem proving through the formalization of *Z*'s syntax and semantics within the host logic (7:160).

2.3.4 ZOLA. ZOLA is a tool for the creation, maintenance, and verification of *Z* specifications. The two modes for entering specifications are a syntax-directed editor and an ASCII parser. Presumably, the former mode enhances familiarity with the *Z* syntax, while the latter mode speeds up the editing process. The tool uses a tactical theorem prover that implements the inference rules of the logic as tactics. The application of these tactics converts the original goal into less complicated goals through instantiation or term rewriting (50:9). The current state of the goals in a proof is stored as part of the specification's state and can be extracted later for re-accomplishing the proof. The soundness of any theorem in a Z specification is ensured through strict control of the specification's environment, and any changes made will invalidate the proven theorem.

2.4 Object Orientation and Z

As discussed in Section 2.2, the *Z* language can specify state and behavior via the schema construct. However, this construct does not provide for the grouping of operations on a particular state, the application of these operations to different instances, or the inheritance of these properties. Object orientation provides a method for structuring

software and can extend *Z* by including support for classes and inheritance. The use of objects divides up a system's state space into meaningful portions, while the inheritance of class instances manages the system's complexity. Object orientation also offers support for structuring large, complex systems through modularity, and for reuse of specifications and proofs (5:269). Since the inadequacy of *Z* in these areas has been well documented, the pursuit of these benefits has produced two different approaches for structuring *Z* within an object-oriented mechanism: *Z* in an object-oriented style (object-based) and object-oriented extensions to *Z*.

2.4.1 Object-Based Usage of Z.

2.4.1.1 Hall's Style. The standard *Z* language is capable of producing descriptions of object-like entities that have states and operations, but it cannot distinguish the individuality or identity of each "object". Hall's Style of *Z* proposes a change to the way that basic state and operation schemas are expressed, whereby the state of the individual objects is extended to include an explicit notion of self (59:35). Different values of the self variable represent the different identities of individual objects. For example, the state of a quadrilateral is represented in Figure 2.1 (51:20):

```
─ Quad ──────────────────────────
│ self : QUAD
│ edges : Edges
│ position : VECTOR
└────────────────────────────────
```

Figure 2.1 Hall's State Schema

The delta state operation is also extended to ensure that the self of the quadrilateral object remains constant, as shown in Figure 2.2.

$$
\begin{array}{|l}
\hline
\textit{QuadOp} \\
\Delta Quad \\
\hline
self' = self \\
\hline
\end{array}
$$

Figure 2.2 Hall's Operation Schema

This approach gives an explicit model for the collection of objects that comprise a system. The system stores the mappings from object identities to the corresponding object states, and the predicates ensure that the mappings are correct. For the purists, this Z object-oriented method is acceptable, since only the standard language is used. However, the flatness of the standard language does not support the object-oriented notion of a class, thereby limiting object compositions and also prohibiting the use of inheritance (59:38).

2.4.1.2 Z Expression of Refinable Objects (ZERO). This methodology was developed to address the difficulties that inhibit the refinement of large, formally specified systems. ZERO reduces the level of interaction between objects in a large system by introducing additional structuring into the specifications. Without significantly altering the standard Z notation, all objects are described separately by export and body specifications in order to permit reasoning about them without considering any internal details. The export specifications algebraically describe the object's behavior, while the body specifications describe its state and methods (61:29).

ZERO includes intuitive mechanisms for representing standard object-oriented notions such as inheritance, instantiation, and invocation of objects (59:207). These mechanisms make it possible for an operation schema to refer to another object by using that object's export as a type and then invoking methods on that object. This facilitates reasoning about an operation's results without expanding the specifications to inspect their signatures. Unfortunately, this approach demands greater proof obligations between the export and body specifications, and it has not been proven that this definition of export specifications will result in a solution.

2.4.2 Object-Oriented Extensions to Z. This approach to combining object orientation and Z has resulted in many different variations on the theme of wrapping the language within a structured framework. The semantics of each approach are significantly different, but the syntax is generally consistent. The common features found in most of these object-oriented extensions are (29:81):

- The standard Z is extended to allow the definition of classes, which permit the definition of the structure and behavior of similar objects.
- Classes can be extended through inheritance.
- Operations can be polymorphically defined.
- The objects communicate through message-passing.

The following sections provide an overview of the features, syntax, and application areas of four Z extensions: Object-Z, Z^{++}, OOZE, and Moo$Z$. These particular examples

were chosen based on their relative maturity of development and application to substantial case studies.

2.4.2.1 Object-Z. Object-Z is an extension to the language to conform to the constructs of object orientation. In addition to the standard features such as encapsulation and inheritance, this language also supports instantiation, aggregation, and message-passing. Most importantly, however, Object-Z improves the clarity of large specifications by addressing the problem of relating multiple state schemas to one operation schema (38:59). In standard Z, determining which operation schemas affect a particular state schema can only be accomplished by examining the signatures of *all* operation schemas. Object-Z overcomes this dilemma by introducing a new schema type called the "class", which acts as a template for all objects of that class. Each object's states are instances of the class' state schema, and its individual transitions correspond to individual operations of the class (39:11). A complex class can inherit other classes, but each individual class can also be considered separately. The template for an Object-Z class definition is detailed in Figure 2.3.

Within this template, the visibility list, if included, restricts access to the listed attributes and operations of objects of the class. If the visibility list is omitted, then all features (the collective attributes and operations) are visible. In this language, the state schema is an unnamed Z schema which declares the attributes of the class, while the operation schemas use Z schema notation to define state transitions. The history invariant is a predicate over histories of objects of the class, thereby restricting their behavior.

ClassName [generic parameters]

visibility list
inherited classes
type definitions
constant definitions
state schema
initial state schema
operation schemas

---

history invariant

Figure 2.3 Object-Z Class Definition

An evaluation of the maturity of the language reveals that Object-Z has a well developed theory of temporal logic specification, but it still lacks sufficiently well-defined semantics for sequential specification aspects. A theory for refinement of the language and techniques for transformation to code (C^{++}) have been developed, but the model of time is still discrete rather than real-time. Also, the language lacks an integrated approach to the use of informal notations with the formal language notation. However, despite the drawbacks, Object-Z is considered to be the most widely accepted object-oriented extension to the standard Z language, based on the number of applications in the language (19:26).

2.4.2.2 Z^{++}. The Z^{++} language was developed as a notation that could naturally express designs and also as a means for correctly structuring large system specifications. Its syntax supports temporal logic and is therefore quite unique; it is different from both standard Z and the other object-oriented extensions to the language (20:138). A Z^{++} specification can be composed of a sequence of Z paragraphs or Z^{++} class definitions, which are more restrictive. Additionally, the language permits class names to be used as

types, while class methods can be used as operations or schemas (21:105). A description of a Z^{++} class declaration is detailed in Figure 2.4.

Object_Class ::= CLASS Identifier TypeParameters [EXTENDS Imported]

[TYPES Types] [FUNCTIONS Axdefs]
[OWNS Locals]
[RETURNS Optypes]
[OPERATIONS Optypes]
[INVARIANT Predicate]
[ACTIONS Acts]
[CONSTRAINTS Constraints]
[HISTORY History]

END CLASS

Figure 2.4 Z^{++} Class Specification

The semantics of Z^{++} are algebraically-based, making it possible to algebraically prove properties about a specification. Therefore, category theory building operations such as product and co-product can be applied to the defined classes to achieve refinement. Additionally, a model-based theory provides a method for reasoning about the classes and their refinements. Some deficiencies of Z^{++} include the forbidding of circular references between classes, the lack of object self-reference, and the inability of one object of a class to refer to another object of the same class (19:28).

2.4.2.3 Object-Oriented Z Environment (OOZE). OOZE is a programming environment that uses the graphical notation and comment conventions of Z, formalizes its style, and adapts the language to fit the object-oriented paradigm (2:79). This modification is based on an order sorted algebra (vice the first order logic and axiomatic set theory of Z), which simplifies reasoning and mechanical verification of specifications. Since the

semantics are similar to those of the algebraic specification language OBJ3, it is possible to use term rewriting to execute certain OOZE specifications. The overall environment includes a syntax checker, a type checker, an interpreter for an executable sub-language, a theorem prover, and a module database. The use of generic modules enforces the notion of encapsulation through the hierarchical organization of specifications.

In this language, a clear distinction is made between objects, or instances, and class declarations, which serve as templates for the objects. The general form of an OOZE class specification is shown in Figure 2.5. In this methodology, the *initial values* attribute denotes the initial state of the class, while *methods* defines the operations that can affect the attributes of a class. These methods are written in the conventional *Z* schema notation, and each operation is constrained by a precondition, which must evaluate to true in order for the state variable updates to occur (19:24).

Class class-name < *ancestor-names*

constants

State
class attributes
class invariant

Init
initial values

methods

Figure 2.5 OOZE Class Specification

The OOZE language contains another structuring concept called the module. A module is used to encapsulate all classes that have interdependent relationships, or to

group related specification components. A particular form of module, called theories, provides a method for stating the properties of class parameters. Theories can also be parameterized in addition to using and inheriting other theories. The general form of a theory module is seen in Figure 2.6.

```
┌── theory-name ──────────────────────────
│
│ types
│
│  │ axiom-declarations
│  ├──────────────────
│  │ axioms
│
└─────────────────────────────────────────
```

Figure 2.6 OOZE Theory Description

OOZE is considered to be a "wide spectrum" object-oriented language based on its loose specifications and executable programs (2:94). The use of loose specifications is generally regarded as a powerful semantic type system, and executable prototyping permits the demonstration of the specified behavior. These characteristics of OOZE, coupled with support for re-usability and the reduction of complexity, make this environment well-suited for the formal-based development of large software systems.

2.4.2.4 MooZ. The goal of the Moo$Z$ language is to provide support for the object-oriented design and management of very large specifications. It is considered to be similar to Object-Z, although its semantics are actually much simpler. A Moo$Z$ specification is a set of hierarchically related classes with objects as values whose types are classes (29:80). A bottom-up structure is inherent in the language based on classification

and inheritance mechanisms. All subsystems are properly encapsulated, clarifying the behavior of whole systems as the interaction of its defined classes.

The general outline of a Moo$Z$ class is depicted in Figure 2.7. Some unique features of this definition include the *givensets*, *state*, and *operations* attributes. Similar to "given sets" in standard Z (45), the facet *givensets* declares sets whose existence is assumed to be present, but no details are defined. In Moo$Z$, this feature permits the parameterization of classes. The *state* facet describes the attributes of the state of a class and an invariant or predicate expected to be true for these variables between invocations. The state is described by an anonymous state schema written in standard Z. The *operations* facet, also written in standard notation, describes a set of methods that transform the state of the class.

Class <*Class-Name*>
givensets <*type-name-list*>
superclasses <*class-reference-list*>
<*auxiliary-definitions*>
private <*definition-name-list*>
or
public <*definition-name-list*>
constants <*axiomatic-description-list*>
<*auxiliary-definitions*>
state <*anonymous-schema*> *or* <*constraint*>
<*auxiliary-definitions*>
initialstates <*schema*>
<*auxiliary-definitions*>
operations <*definition-list*>
EndClass <*Class-Name*>

Figure 2.7 Moo$Z$ Class Definition

Moo$Z$ uses a variation of Spivey's Z Mathematical ToolKit, where each type constructor is defined using a generic class instead of a set, thereby representing mathematical

definitions as objects (28:42). A special superclass called *MathematicalDefinitions* is defined over all other Moo$Z$ classes, permitting easy access to all of the ToolKit definitions. Other distinguishing features of Moo$Z$ are the representation of objects as records and the treatment of schemas as messages. In comparison to the other Z extensions, Moo$Z$ adheres more to the object-oriented paradigm than to the semantics of the Z language.

2.5 Integrating Z and Structured Development Methods

The continuous increase in the size and complexity of many software systems has created a demand for more precise methods of capturing and specifying their desired behavior. As a result, many different types of specification methods now exist. Formal languages, such as Z, are concise and unambiguous notations that permit reasoning but lack structure for management. Alternatively, structured development methods, such as structured analysis or object-oriented analysis (OOA), provide a framework for managing the size and complexity of the specification, but they lack formality. The integration of these two methodologies capitalizes on the individual advantages and produces software specifications that are both formal and problem-oriented (43:229).

One example of an integrated methodology is AFIT's formal extension of Rumbaugh's OOA model (40) using Z (17). This approach is detailed in Chapter III based on its selection as the starting point for the formalized object transformation process described in this thesis. As a comparison, a brief description of a second approach combining Z and structured development methods is presented in the following section.

2.5.1 Yourdon and Z. One example of an integrated methodology that combines formal notation with structured software development techniques is the extension and formalization of Yourdon's structured analysis approach using the Z notation (43:228). This two-phase method uses the entity relationship diagram (ERD) as the basis for deriving the Z state schemas, while the semantics of each of the processes in the data flow diagram (DFD) are specified using Z operation schemas. This analysis produces an abstract, formal specification of both the static and dynamic properties of the target system.

The first phase, depicted in Figure 2.8, produces a data model of the system being developed. This data model is initially expressed as the combination of an ERD and any associated attribute lists. Rectangles are used to represent entities, and diamonds are used to represent relationships. The use of the graphical ERD notation as the basis for the development of a Z specification is well-founded, since the diagrams have a straightforward interpretation in terms of set theory.

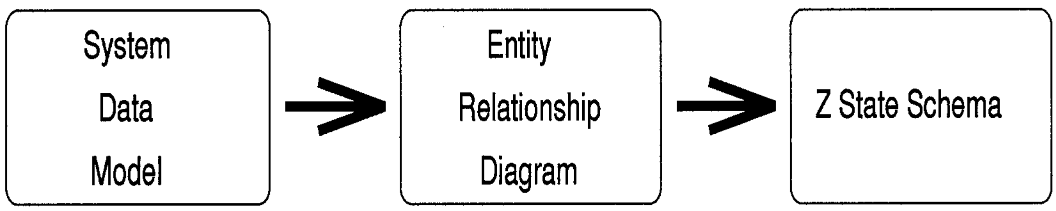

Figure 2.8 Yourdon Derivation of State Schemas

Once the ERDs have been completed, the discrete process of deriving the Z state schemas is accomplished in four steps:

1. Define a basic type for each attribute, and then define a schema type to represent the entity.

2. Define state schemas for the instances of each entity and any subtypes. (Including associative entities.)

3. Declare relations to model the simple named relationships in the ERD.

4. Combine the entity and relationship schemas to provide a complete specification of the system state, adding predicates to constrain any relationships.

The second phase, depicted in Figure 2.9, produces a process model that describes the dynamic behavior of the system. First, a semi-formal model is expressed using DFDs composed of input and output data flows, processes, and data stores. The data flows are represented as labelled arrows, the processes are circles, and the data stores are parallel lines. The DFDs now represent the processing requirements of a specific system as a set of data transformations; however, since the DFDs do not explicitly capture what each process actually does, it is necessary to provide a specification of each transformation's processes. Using Z, it is possible to write these process specifications in an abstract, yet correct manner (43:238).

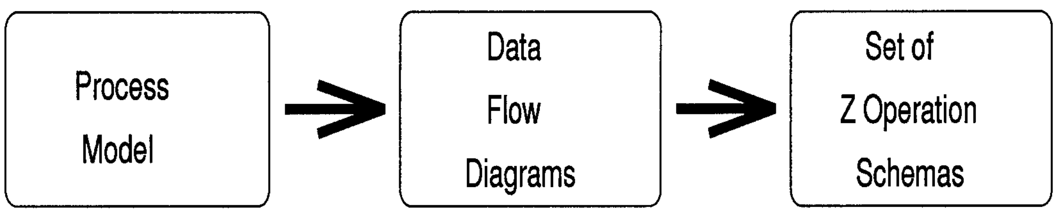

Figure 2.9 Yourdon Derivation of Operation Schemas

First, all the data in the system is collected in the data dictionary, and the exact composition of the data flows is specified. Then each identified process is expressed as a set of one or more Z operation schemas according to the following guidelines:

1. Use the Ξ convention if the state is unaffected (i.e. there are no flows to any data stores).
2. Use the Δ convention if there are any flows to the data stores.
3. Any data stores not written to can be included in the predicate with the Θ prefix.
4. The inputs and outputs to each operation schema can be obtained from the content of the data flows.
5. The relationships between inputs and outputs and state changes are not defined by the DFD and must be specified as preconditions to complete any operation schemas.

The result of this integrated methodology is a formal specification of both the static and dynamic properties of a system. This approach retains the usability of structured analysis and incorporates a mathematically correct notation that is both precise and unambiguous. It is currently limited in application to information systems only, but extensions for dealing with concurrency are being developed. Additionally, the developers have adapted their method to follow an object-oriented approach consisting of an information model (ERD) and the corresponding state models (state transition diagrams (STD)). The overall system is then regarded as a network of communicating state machines.

2.6 Algebraic Specification Languages

In Chapter I, two perspectives of formal specification generation were introduced: model-based and theory-based. An example of the former perspective is the *Z* language, which builds an abstract model of a system's states and operations. The contrasting theory-based approach is represented by algebraic specifications, which characterize a system by

describing its properties (49:104). These languages are based on the concept of defining an abstract data type (ADT) as a many-sorted algebra, which comprises one or several sets of values, called sorts, and operations on these sets (15:3). These sorts and operations provide the basis for building a theory, which captures three distinct facts about the ADT:

1. Signature - A listing of the sort names, operation names, and the applicable domains and co-domains of these operations. The signature forms a domain vocabulary for the specification.

2. Axioms - The logical formulas generated by the signature's vocabulary. The axioms constrain the operations and must be satisfied by the underlying algebras. Derivations from the types of axioms used, e.g., equations, Horn clauses, or first-order logic, determine the *presentation* of the theory (49:94).

3. Models - These are the algebras denoting a set for each sort and a function or relation for each operation. The behavior of these models must satisfy the specified axioms.

The variety of logics used for defining axioms has produced many different algebraic specification languages. Some of the more representative languages and their underlying logics and semantics are described in the paragraphs below.

2.6.1 ACT ONE. ACT ONE is a kernel specification language, restricted to basic algebraic data type specifications and their structuring mechanisms. Coupled with equational logic and denotational semantics, these factors actually limit descriptive power and prohibit "loose specifications", that is, multiple variable bindings that satisfy global specification properties (62:270). Its theory building operations include combine, rename, and actualize. The combine operation produces the union of specifications and permits the

sharing of data when used in conjunction with the renaming operation. The actualization operation provides for the instantiation of parameters. ACT ONE supports a structuring methodology based on hierarchy and abstraction. Hierarchies are realized by the extension of the ADTs, while stepwise refinement is used to add details to the original abstraction (11:370). Since ACT ONE is limited in its ability to handle large specifications, an extension to the language is under development. The successor, ACT TWO, supports both horizontal structuring through interconnection of modules, and the vertical structuring of systems.

2.6.2 CLEAR. CLEAR also uses equational logic but instead couples it with loose semantics. This makes the language more robust than ACT ONE as theories may be shared while the building operations remain independent. More complex theories can be built either through the coproduct of shared theories or the enrichment of an existing theory with new sorts, operations, and equations (48:97). CLEAR can also force the particular interpretation of a theory through the derive operation. The power of the language's semantics is based in a body of mathematics called category theory (62:271).

2.6.3 OBJ. OBJ is an executable algebraic specification language that is based on order-sorted logic and both denotational and operational semantics. The executability of the language forces the extra constraints of termination and uniqueness on any rewrite rules (62:271). An OBJ specification has a hierarchical structure whose top-level entries include objects, theories, and modules. The language uses an IMAGE construct as a means of instantiating reusable generic objects from an original object. An object introduces one

or more sorts represented by various identifiers that denote the carrier set of the sort and operations and their signatures.

2.6.4 LARCH. The LARCH language is based on equational logic and predicate calculus for its semantics (62:272). It is uniquely designed to take a two-tiered approach to specification through a shared language and a set of interface languages. The LARCH shared language (LSL) describes objects by using theories that are free of implementation details. Additionally, the LSL is accompanied by a library of specifications and specification components, which define many basic concepts, e.g., stacks and sets. The set of interface languages permits the enrichment of shared language theories to reflect final implementation structures. An interface is specified using a syntax that parallels the target programming language.

2.7 Implementation of Specifications

The construction and implementation of reliable and *correct* software systems from their specifications is often complex and imprecise. In program development, the design stage of the software lifecycle bridges this gap between specification and implementation through various transformations or refinements that incorporate details in manageable increments. The increments can be formalized into frameworks that enforce constraints throughout the transformation, thus incorporating a correctness-preserving quality. Two examples of specification implementation approaches are discussed below.

2.7.1 Transformational Programming. The first approach to the implementation of formally specified software is the integration of formal methods with transformational

programming. This methodology of program construction is based on successive applications of *transformation rules* to a formal specification in order to produce an executable program (33:201). Transformation rules are partial mappings between the various versions of the program such that the relationship between an element of the domain and its image constitutes a correct transformation. This correctness-preserving quality is achieved through predicates that constrain the program scheme.

Transformation rules can be represented in two ways: procedural rules or schematic rules. Procedural rules are often referred to as global or semantic rules and consist of flow analysis, consistency checks, and representations of programming techniques, e.g., divide and conquer (33:203). Schematic rules are typically used in connection with local rules and perform the following functions:

1. Relate language constructs (syntactic rules).

2. Describe algebraic properties relating different language constructs.

3. Express domain knowledge as data-type properties.

Additionally, two hybrid rules, FOLD and UNFOLD, are used to represent implementation rules. Their inverse relationship provides great flexibility in program development through re-usability for whole classes of problems (34:9). Therefore, the entire spectrum of transformation rules comprises these three types (procedural, schematic, hybrid), but most systems extend this set through supplemental advice on how to apply them. Some examples of the typical application options and their purposes are listed below (33:204):

- Enabling conditions - guarantee correct application

- Strategic conditions - focus on a specific goal
- Continuation information - suggests rule(s) to try next
- Scope of application - points out other rule possibilities

Transformational programming is used to achieve a variety of distinct software development goals. These goals include general support for program modification, program synthesis, program adaptation, and program explanation (34:11). Additionally, this paradigm supports the fundamental software engineering attributes of efficiency, portability, and reusability. But its biggest contribution is in the area of producing safe, reliable, and correct software from formal specifications.

2.7.2 Refinement. Another approach to the implementation of software specifications is the process of refinement. Refinement is a technique that executes each design decision by incorporating the new information, through a series of formally provable steps, into a new version of the specification. These steps, or *refinement rules*, demonstrate the satisfaction relations between the successive versions of the specifications (60:5). Since any mistakes will be reflected as failures in the satisfaction relation, this type of proof reduces doubt about the specification's validity and verifies its interpretation.

The refinement process adds extra detail to each specification through either data or operation *reification.* The *reification* process is defined as the mapping of abstract information into a more concrete, or machine-processable state. The specific aim of data refinement is to produce *adequate* representations of all the abstract types within the specification. Adequacy is determined by the representative power of the concrete syntax; every value in the abstract type must be retrievable from the concrete type (60:7). Additionally,

data refinement addresses which operations will access the data and establishes mappings to be used during operation refinement.

While data refinement deals with the passive state of the system, operation refinement concerns the active part of the system. The principle of proving that the concrete operations do the same thing as the abstract ones is known as *satisfaction.* This satisfaction constraint of operation refinement is complementary to the adequacy constraint of data refinement. Operation refinement can either augment implicit information in the specification or implement algorithms capable of producing the required postconditions or resultant properties.

There are two basic approaches to refinement, either by design and proof or refinement by transformation. In the first approach, the verification of correctness is accomplished through a derivation of proof obligations that are subsequently discharged. Interactive theorem provers such as HOL can be used for the refinement proof system, but little work has been accomplished in the area of generating proof obligations (60:14). Variations on this approach also include specification type checkers and animators, which do incorporate some degree of a correctness-preserving quality. The second approach, refinement by transformation, has been automated and is supported by various tools, e.g., Software Refinery. The Software Refinery development environment provides a domain for specifying state transformations through descriptions of a precondition and a postcondition. Additionally, the REFINE wide-spectrum language provides constructs to implicitly introduce the appropriate control structures when required.

Refinement is a viable solution for the complexity and imprecision of program development. Since the technique uses a series of formally provable steps to implement specifications, it provides some degree of correctness-preservation; however, although it is based on a well established theory, refinement does not scale easily to large specifications nor does it adequately handle a requirement's functional constraints (60:17). More work is needed to address these drawbacks and expand the applications of this technique.

2.8 Conclusion

This compendium of information established a three-pronged knowledge base that forms the foundation for this thesis. First, in order to construct a viable *Z*-based object transformation process, the design methodology should address issues of coverage, correctness, and adherence to an accepted standard form of the language. Second, in terms of executability, an incremental approach provides manageability and facilitates constraint enforcement, thereby preserving correctness. Third, the feasibility of constructing an accurate unifying domain model between *Z* and a selected algebraic specification language is realized by their common core of fundamental mathematical structures.

III. *Design of a Formalized Object-Based Transformation Process*

3.1 Introduction

In Chapter I, the notion of the next generation application composition system was introduced and an integral part, a formalized object-based transformation process, was depicted in Figure 1.1. A major goal of this future composition system is the ability to construct large systems out of smaller, well-founded pieces while formally mapping between levels of abstraction (4). Consequently, the design and development of an internal transformation mechanism must strive for a product which is flexible, extensible, reliable, and verifiable.

For this research effort, the transformation mechanism is an object-oriented *Z* to REFINE compiler that incorporates a unification with algebraic specification languages. In terms of functionality, the system is partitioned into four distinguishable components: Rumbaugh's OOA framework (40), a *Z* compiler, a unifying model, and a REFINE execution framework. The requirements definition phase identified the projected methods and tools for all of the components except the unifying model; a final parameter, the target algebraic language, was required for completion. Therefore, as a part of her concurrent research, Captain Lin analyzed two specification languages, OBJ and LARCH, for problem suitability and executability. Based on its potential adaptability to the existing OOA framework, availability of support tools, and two-tiered structure, LARCH was selected as the target algebraic language. (A complete discussion of the analysis can be found in Lin's thesis (23).) With the completion of this phase, the design of the specified object transformation process was started. This chapter describes the selected development approach, the design

validation criteria, and problem domains in Sections 3.3 through 3.5 after an overview of the *Z*-based extension to the OOA framework.

3.2 Formal Extension of OOA Using Z

As discussed in Section 2.5, another example of an integrated specification methodology is demonstrated by AFIT's formal extension of Rumbaugh's OOA model (40) using *Z* (17). The similarities to the structured analysis method of Yourdon are demonstrated by a procedural decomposition process and the diagrams that detail the resultant components. Specifically, the Rumbaugh model of a system is composed of three distinct views: an object model, a dynamic model, and a functional model. These views are represented by entity relationship diagrams, state transition diagrams, and data flow diagrams, respectively. As in the previous Yourdon methodology, the three phases of the formal extension using *Z* produce an abstract, formal specification of both the static and dynamic properties of the target system.

The first phase, depicted in Figure 3.1, generates the object model, which identifies not only the objects and their attributes, but their relationships (associations, aggregations, inheritance) as well.

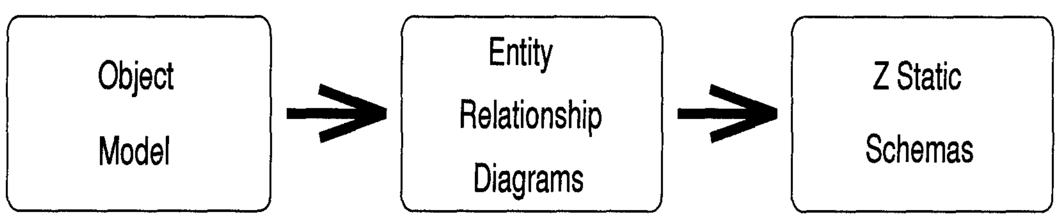

Figure 3.1 Rumbaugh Derivation of Static Schemas

With some slight differences in the ERD symbology used in the Yourdon method, the object model is captured with the graphical notation. The object model is then formalized by defining a *Z* static schema for each object class. The signature of the schema defines a set-theoretic type for each object attribute. The predicate portion of the schema contains the invariants on and between the object's attributes, as well as any derived attributes (17:12). To illustrate the object transformation process, Figure 3.2 depicts an object model for a Traffic Light class while the corresponding *Z* schema is shown in Figure 3.3.

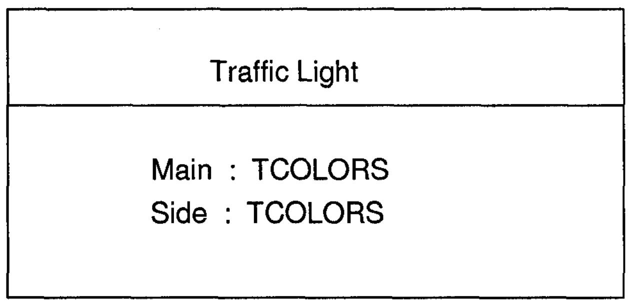

Figure 3.2 Traffic Light Object Model

$TCOLORS ::= Red \mid Yellow \mid Green$

$$\begin{array}{|l}
\hline
\textit{TrafficLight} \\
Main : TCOLORS \\
Side : TCOLORS \\
\hline
Main = Red \lor Side = Red \\
\hline
\end{array}$$

Figure 3.3 Traffic Light Static Schema

The second phase, depicted in Figure 3.4, begins with the dynamic model and produces a set of state transition diagrams that captures an object's individual states and its overall state space.

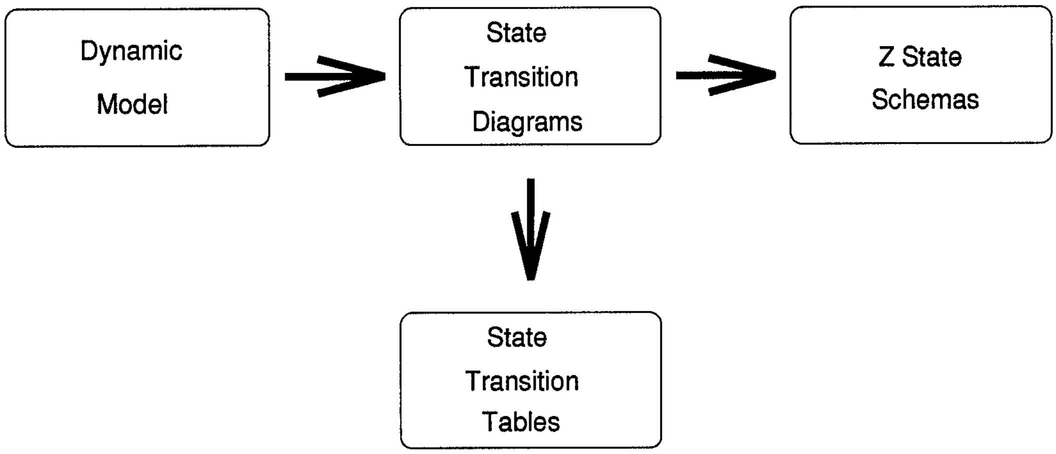

Figure 3.4 Rumbaugh Derivation of State Schemas

For the derivation of Z state schemas, the following steps are accomplished:

1. Name each state and define a corresponding Z state schema by declaring an object of the specified type and including any associations.

2. Declare the partition of applicable state variables using tuple notation and/or membership in the appropriate associations.

3. Include a narrative description of the behavior of an object while in a state. (This information will be expanded in the functional model.)

4. Use the information from the Z schemas to create a state transition table to formally define all the transitions diagrammed in the STD.

5. For each object, generate a partial event flow diagram to document the lines of communication with other objects.

Continuing with the Traffic Light example, the following figures represent the dynamic transformation process: Figure 3.5 contains the dynamic model, Table 3.1 depicts the state transition table, and Figure 3.6 specifies the respective state schemas.

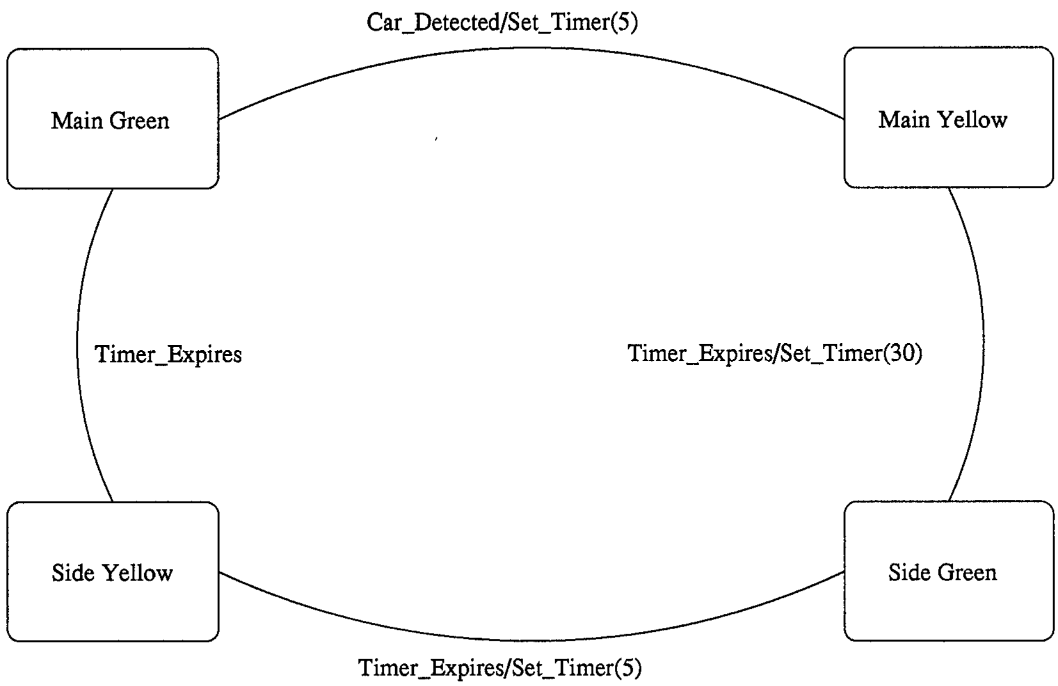

Figure 3.5 Traffic Light State Transition Diagram

Table 3.1 Traffic Light State Transition Table.

Current	Event	Guard	Next	Action
MainGreen	Car_Detected		MainYellow	Set_Timer(5)
MainYellow	Timer_Expires		SideGreen	Set_Timer(30)
SideGreen	Timer_Expires		SideYellow	Set_Timer(5)
SideYellow	Timer_Expires		MainGreen	

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MainGreen}} \\ \hline
\textit{TrafficLight} \\ \hline
\textit{Main} = \textit{Green} \\
\textit{Side} = \textit{Red} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MainYellow}} \\ \hline
\textit{TrafficLight} \\ \hline
\textit{Main} = \textit{Yellow} \\
\textit{Side} = \textit{Red} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MainRed}} \\ \hline
\textit{TrafficLight} \\ \hline
\textit{Main} = \textit{Red} \\
\textit{Side} = \textit{Green} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SideYellow}} \\ \hline
\textit{TrafficLight} \\ \hline
\textit{Main} = \textit{Red} \\
\textit{Side} = \textit{Yellow} \\ \hline
\end{array}$$

Figure 3.6 Traffic Light State Schemas

The final phase of this approach, depicted in Figure 3.7, describes the operations defined in the object model and the actions defined in the dynamic model. As in the Yourdon method, the generated DFDs capture the functional behavior of the object through required calculations or processes, including the sources and sinks of all data. Z operation schemas are then defined for each leaf process. If an object's attributes are unmodified by the operation, the Ξ convention is used, while conversely, the Δ convention is used for modified attributes.

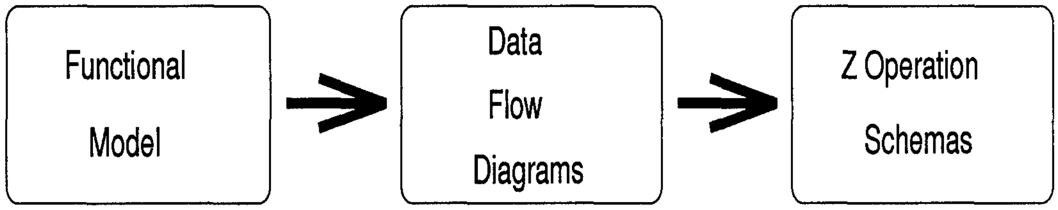

Figure 3.7 Rumbaugh Derivation of Operation Schemas

Completing the Traffic Light example, Figures 3.8 and 3.9 illustrate a sample functional model and its defined Z operation schema, respectively.

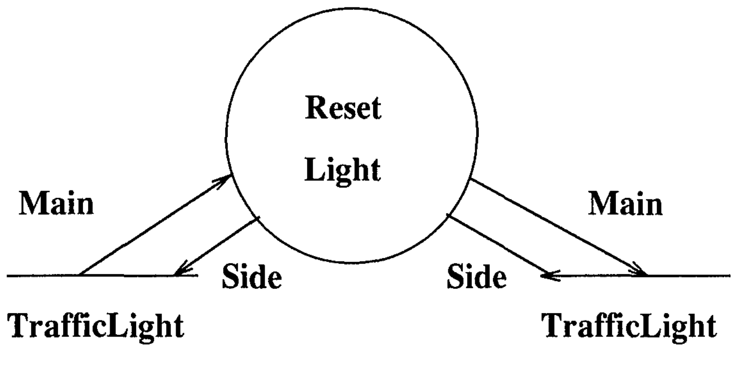

Figure 3.8 Functional Model for Resetting the Traffic Light

ResetTrafficLight

$\Delta TrafficLight$

$Main' = Green$

$Side' = Red$

Figure 3.9 Traffic Light Operation Schema

This integrated development method permits a bottom-up approach of composing formal specifications in an object-oriented framework. As demonstrated, it is similar to the integration of Yourdon structured analysis and *Z*. The one drawback is the incomplete handling of the generated state transition tables, and this is presently being addressed by the developers, Hartrum and Bailor.

3.3 Compilation Process Model

In Chapter I, a major objective of this research was defined as the development and validation of a *Z* language compiler, correlated to the *Z*-based OOA framework. Therefore, the initial step in this design effort focused on defining a baseline model of the compilation process. Standard convention dictates that a compiler is a program, or set of instructions, that reads a program in one *source* language and translates it into an equivalent program in a second *target* language. This compilation process, depicted in Figure 3.10, has two major portions: analysis and synthesis (1:2).

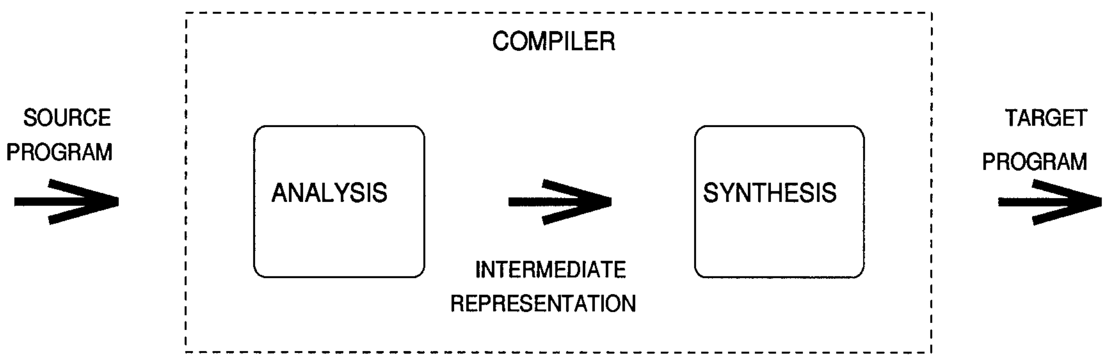

Figure 3.10 Analysis-Synthesis Model

The analysis portion breaks up the source program into its integral pieces and creates an intermediate representation. The intermediate form is a compact version of the source

program that eliminates redundancy and ambiguity while capturing an explicit model of the input. The synthesis portion then constructs the desired target program from this intermediate representation.

Based on its inherent modularity, this process model provides a manageable framework for the object-oriented translation of the abstract source language Z into the wide-spectrum target language of REFINE. The framework was further partitioned via the inclusion of the following constraints within the development process:

- Coverage - the portion of the Z language successfully handled.
- Consistency - the preservation of the original specification as a mathematical object. Three main areas of measurable consistency, i.e., addressable through reasoning tasks, are listed below (35:237):
  1. Consistency of Global Definitions - There exist values for global declarations that satisfy the global predicates.
  2. Consistency of the State Model - There is a state of the general model and applicable inputs that satisfy the initial state description.
  3. Consistency of Operations - The calculated preconditions of the operation schemas do not evaluate to *false*.
- Correctness - the verification of the originally specified behavior.

Once these goals were established, the design effort concentrated on the selection of the appropriate development approach.

3.4 Development Approach

There are many different development strategies for implementing a software system, e.g., evolutionary, water-fall, spiral. Prior to selection, a strategy should be evaluated for project suitability based upon the following factors (52:3-15):

- Nature and application of the program
- Projected methods and tools
- Required controls and deliverables

Since these factors map to the framework and constraints discussed in Section 3.3, the various strategies were assessed against the defined foundation. In the outcome, evolutionary development was deemed appropriate for this research effort. Different than prototyping in terms of time and effort, this strategy involves developing a fully operational and documented initial product and then successively developing more refined editions (52:3-16). The initial capability meets minimum essential requirements and is identified by a flexible, modular structure that can accommodate future changes. Inherent to this strategy are the incremental demonstrations of the product's capabilities, thus incorporating valuable feedback into the development process.

Using this evolutionary approach, the compiler development was separated into three major iterations which are detailed in Chapters IV, V, and VI respectively:

1. Analysis I - Z parser development (including Mathematical ToolKit)
2. Analysis II - Unified model development
3. Synthesis - Execution framework development

In terms of execution, a complete transformation of Z into REFINE was not accomplished. Rather, an execution framework composed of consistent and demonstrable mappings between the source and target languages was developed. The justification for this outcome was directly tied to the development strategy. As previously stated, the fully operational initial product of an evolutionary development approach requires a great investment of time and effort. Since maximum coverage of the Z language was defined as an essential requirement, the development of the parser in the first evolution was indeed slow and meticulous. The added feature of a separate Mathematical ToolKit amplified the complexity of the task, but, more importantly, it produced a tool that adhered to the accepted definition of the language!

3.5 Validation Criteria and Domains

Within this evolutionary design strategy, two validation criteria were established: preservation of the OOA models' properties and preservation of the structure and meaning of the Z language. In order to satisfy these guidelines, three distinct phases were used to provide comprehensive coverage. Initially, using simple examples, informal testing was conducted to assess each product's success against the predefined requirements. For standardization and comparison, a simple, non-reactive counter specification was designated as the common example for both Z and LARCH. For the second phase, integration testing combined the simple test cases into larger and more sophisticated examples. For each language, various representative examples from a wide range of references were used. A listing of four specific Z examples is provided in Section 4.3.1. In the final phase, a feasibility demonstration was conducted to validate the entire formalized object transformation pro-

cess. To ensure complete coverage of the process, a reactive domain model with complete object, dynamic, and functional representations was selected, namely, the OOA rocket example by Hartrum and Bailor (17). For manageability, the smaller fueltank domain was identified as the common benchmark for both languages. For reference, Appendices A, B, and C contain the different validation specifications.

3.6 Summary

This chapter detailed the formulation of an evolutionary development approach used to design a *Z* language compiler that incorporates a unification with the LARCH specification language. Three essential constraints were identified as a foundation for the development process: coverage, consistency, and correctness. These threads of control guide the three development evolutions detailed in Chapters IV, V, and VI.

IV. Z Parser Development

4.1 Introduction

In Section 3.3, a two-phase compilation process model was introduced as a framework for managing the development of a robust compiler for the *Z* language. The two phases, analysis and synthesis, provide even further internal modularity for handling the task's complexity. Commencing with the analysis section, Figure 4.1 depicts a logical apportionment into three distinct phases.

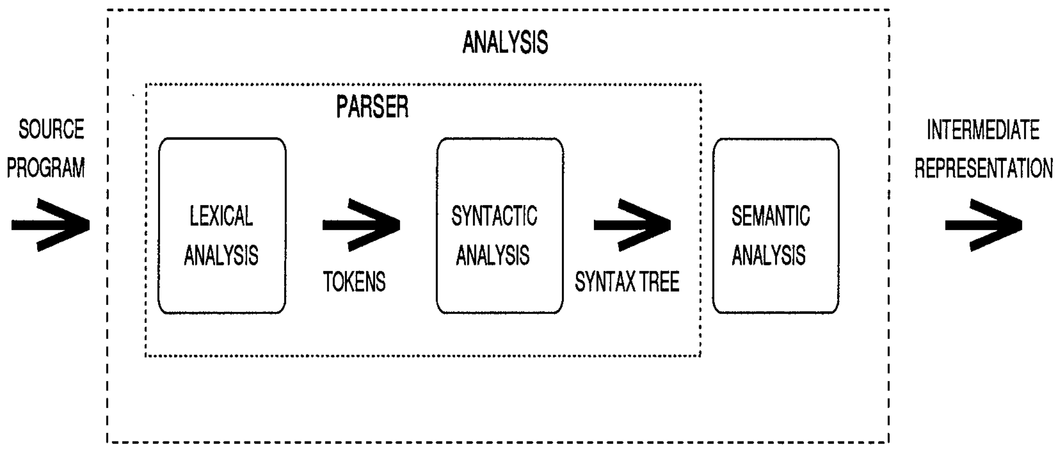

Figure 4.1 Compilation Analysis Phase

The first phase, lexical analysis, scans the source program and groups sequences of characters into tokens that possess a collective meaning. Next, the syntactic analysis phase groups tokens together to form hierarchical syntax trees, which have a recursive basis. As depicted in the diagram, the combination of the lexical and syntactic phases is often called the front-end or "parser". Finally, the third phase, semantic analysis, checks the source program for *meaning* errors and gathers type information for the subsequent synthesis portion of compilation.

This chapter discusses the two major evolutions of the *Z* parser, namely the core language and the Mathematical ToolKit. The analysis, design, and validation of each evolution are outlined in Sections 4.3 and 4.4 after a brief description of the DIALECT component of the REFINE environment.

4.2 The DIALECT *Tool*

The DIALECT tool within Software Refinery constructs an LALR (bottom-up) parser that converts source text into a REFINE object base representation. DIALECT requires two inputs: a source language domain model, which defines the structure of the abstract syntax tree, and a *grammar* written in a special high-level language that is an extension of Backus Naur Form (BNF). BNF is a formalism used to describe the syntax of a language through a sequence of rules (42:3). A rule consists of a left-hand side (LHS) and a right-hand side (RHS), separated by a colon. The LHS consists of a single, unique non-terminal symbol. The RHS consists of a sequence of one or more formulations, which are composed of non-terminal and terminal symbols. DIALECT extends the BNF in two ways: it allows regular expressions to be used in the grammar productions, and it allows the names of object attributes to appear in place of nonterminals in the productions (36:5-1). The syntax of a production takes this form:

$$\langle \text{ nonterminal-name } \rangle ::= [\langle \text{ syntax-for-class } \rangle] \langle \text{ action } \rangle$$

This *grammar* can also include precedence and associativity information for removing ambiguities. Some of the features of the DIALECT grammar's syntax include (36:5-3):

1. Reserved words or symbols (literal text) enclosed within double quotes.

2. Optional constructs enclosed within curly braces.

3. Set-valued or sequenced-valued attributes followed by one of three regular operators: *, +, and ++. Respectively, these symbols represent the number of elements within the set or sequence: zero or more, one or more, two or more.

The domain model defined for the source language is composed of object classes and the relationships defined between them. The parser created by DIALECT uses this domain model to directly construct an abstract syntax tree (AST) without the explicit specification of tree building instructions. Each node in the AST represents a source language object class (the left side of a grammar production rule). The branches of the AST represent the constructs from the right side of the production rules. These branches are annotated with the name of a REFINE structural attribute mapping, which is a relationship between two objects in the domain model (37). Additionally, annotation attributes such as boolean values, can be defined for specific objects as well.

4.3 Z Core Language Parser[1]

4.3.1 Domain Analysis. In order to define an accurate domain model of the core *Z* language, the structure of the language was analyzed in great detail. As the primary reference, the 1989 version of Spivey (45) was used extensively, based upon its recognition as the *de facto* standard in the *Z* community and its accessibility in this research effort. In order to validate the coverage capability of the parser, a variety of specification documents

[1]The original *Z* language and Mathematical ToolKit domain models and grammars described in the following two sections are not included in this document. The final versions, produced during the unification phase (see Chapter V), are detailed in Appendices D - I.

were identified as sample inputs. These examples provided an extensive sampling of Z language constructs and their myriad combinations. Specific documents and their respective sources are listed below:

1. A Data Dictionary by Jia (18)
2. An Aircraft Passenger Listing by Lightfoot (22)
3. A Car Radio by McMorran (27)
4. A Computer Network Online Monitor by Ratcliff (35)

In terms of structure, a Z specification document consists of a sequence of paragraphs containing formal mathematical text interleaved with natural language comments. The various paragraphs gradually introduce the variables, types, and schemas of the specification by building on any preceding definitions. This principle of "definition before use" strengthens the vocabulary of the specification and extends the scope of each element from its definition to the end of the document (45:49). As a standard, the general format of a Z document is as follows (22:52):

- Introduction
- Declaration of types
- The state of the system and its invariant properties
- An initial state
- Operations and enquiries (Δ and Ξ conventions, respectively)
- Error handling
- Final versions of operations and enquiries

4.3.2 Correlation to the OOA Framework. Section 3.2 discussed AFIT's formal extension of Rumbaugh's OOA model using *Z*. In order to correlate the parser to the *Z* used within the OOA framework, all paragraphs were classified into two major subtypes: definition and schema. Definition paragraphs contain type information, axiomatic and generic definitions, global constants, and schema calculus expressions. This type of paragraph was distinguished from the intrinsic schema paragraphs because they provide supplemental information to Rumbaugh's object, dynamic, and functional models.

Schema paragraphs are categorized into four distinct types. In addition to the three types of schemas (static, state, and operation) described in Section 3.2, the *Z* parser was designed to recognize *event* schemas that correspond to events contained in the system's state transition table(s). Although standard *Z* does not differentiate between schema types at the syntactic level, the declaration of individual schema types in this parser resolved ambiguity problems encountered by DIALECT.

Each type of schema paragraph was defined to have two major components: a schema name or id and a schema body construct, which is composed of a declaration part and an axiom part. Some specific attributes of these components are as follows:

1. Schema name - A variable used to designate a schema. Once introduced, this id becomes global to the specification and cannot be used in any other context.
2. Declaration part - This portion of the schema introduces one or more variables that are composed of a name and a type. It is a required construct in all of the schema types except event schemas. The declaration part can also contain references to other schemas in the form of inheritance, inclusion, or Δ/Ξ conventions (see section 4.3.3).

3. Axiom part - This portion of the schema is required in the event schema (default value of true), but optional in all others. When present, it contains one or more predicates that describe properties or constraints of the schema variables. The predicates ensure that *meaningless* states of the modeled system are not permitted.

Figure 4.2 illustrates the hierarchical structure of the schema paragraphs:

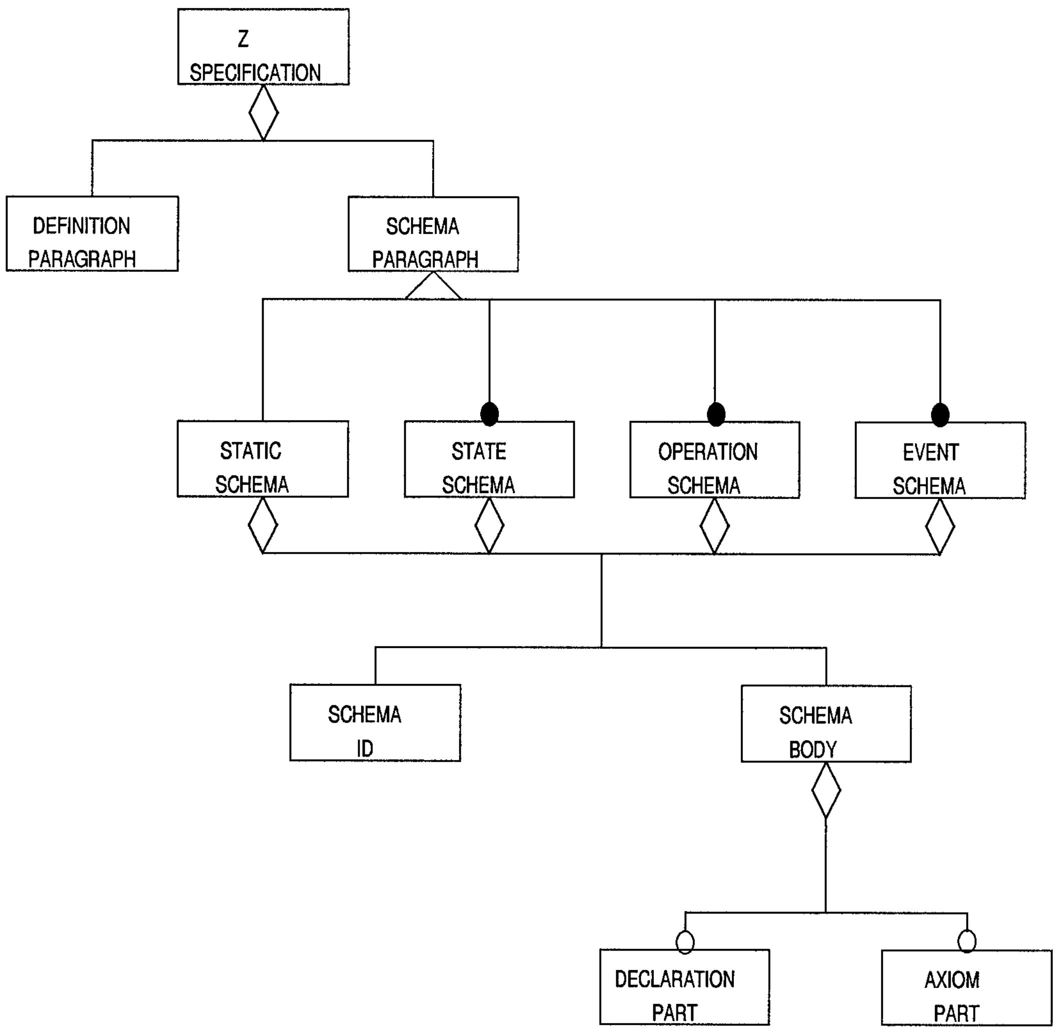

Figure 4.2 *Z* Schema Components

4.3.3 Major Object Classes. In addition to the various types of paragraphs described above, the *Z* parser also processes the following major object classes integral to a specification document:

1. Schema references - This construct permits one schema to *import* the text of another schema by referencing the latter's name in its declaration part (35:40). There are four distinct methods for schema referencing: inheritance, inclusion, and the Δ and Ξ conventions. Since the actual mechanics and meaning of each type of import are unique, individual object classes were declared to remove ambiguity during the future semantic analysis phase. A more in-depth discussion of schema references can be found in Section 6.2.1.

2. Decorations - There are three standard decorations used for describing operations on abstract data types: the tick (') for labeling the final state of an operation, the question mark (?) for labeling inputs, and the exclamation point (!) for labeling outputs. Once again, the distinct meaning of each decoration led to the definition of individual object classes to clarify any ambiguities during future semantic analysis.

3. Symbols - Four subtypes of symbols were identified as object classes: infix, prefix, postfix, and miscellaneous. In all cases, except for the infix operators "$=$" and "$\in$", only the applications of the symbols are defined in the core language, while the actual symbols are defined in the Mathematical ToolKit (See Section 4.4). Additionally, the core *Z* language has a variety of symbols "built-in" as basic elements. These are classified as the operators of propositional logic and the schema calculus and are explicit components of the remaining three object classes.

4. Schema calculus expressions - Schema calculus permits short-hand, structured descriptions of schema combinations. There are four broad categories of combining operations: logical ($\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$), quantifying ($\forall$, $\exists$, $\exists_1$), hiding ($\setminus$, $\preceq$) and special purpose, such as calculating preconditions and sequential composition ($\fatsemi$).

5. Predicates - Predicates are very similar to the schema calculus expressions in their usage of the logical ($\neg$, $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$) and quantifying ($\forall$, $\exists$, $\exists_1$) operations. Additionally, predicates can use infix relation symbols or be simply "true" or "false". A specifier is provided the capability for distinguishing between precondition and postcondition predicates via the LaTeX "also" command. The pretty-printed output separates the two types with a section of whitespace and the AST attributes are labeled accordingly for further semantic analysis.

6. Expressions - The core *Z* language has four subtypes of expressions, which range from tuples, bags, and sets, to λ and μ expressions. Some of the other "built-in" symbols include Power set ($\mathbb{P}$), Cartesian product ($\times$), and binding formation (θ).

4.3.4 Core Language Grammar. As a prerequisite for building the core grammar, a "style" was designated for input files to the compiler. There are many *Z* style options available, and their usage is dependent upon the target operating environment and desired output format. Since the LaTeX typesetting software is widely used in the AFIT environment, its **zed** style option was selected as the desired notation for all input files (47). As an additional justification, the LaTeX files required only minor changes to interface with the REFINE environment. The two drawbacks to this choice were the tremendous amount

of commands and mnemonics required to express the Z language, and the work required to map these notations to their corresponding syntactic categories in an existing BNF.

4.3.5 Validation. As stated in Section 3.3, three constraints were levied upon the compiler development process: coverage, consistency, and correctness. In order to adhere to these constraints, the validation of the Z parser was accomplished in the following three phases:

1. Grammar Optimization - This phase consisted of resolving either shift/reduce or reduce/reduce ambiguities reported during the compilation process. Shift/reduce errors occur when the parser cannot determine whether to match the next symbol in one production (shift) or to reduce a string to the start symbol of another production (36:7-2). Reduce/reduce errors are generated when the parser cannot decide which of several reductions to perform. Resolution options included such actions as precedence re-ordering, addition of parentheses, and declaration of individual productions instead of generic ones.

2. Parsing of Source Programs - In addition to the counter and fueltank validation domains, a variety of comprehensive inputs (see Section 4.3) were parsed as source programs to the compiler. Since the core language parser handles a minimal number of expressions and symbols, only an applicable subset of these specifications was tested. The main objective of this phase was to perform a consistency check via a comparison of the resultant programs to the originals.

3. AST Evaluation - Each generated AST was compared to the structure defined in the language's domain model. This process was especially useful in locating missing attribute maps, which created disjoint tree sections.

4.4 Mathematical ToolKit Parser

The second major evolution of the compiler development process was the addition of the Mathematical ToolKit to the validated core language parser. The ToolKit provides simplistic data types that can be used to describe and reason about many different information systems (45:86). The versatility of the standard Z language is based on this library of mathematical definitions implicit within every Z specification (see Section 2.2). The preservation of this underlying relationship was deemed to be the goal of this evolution. DIALECT directly supported this goal through a feature that provides a grammar inheritance mechanism between a common base language and variant dialects. The grammar inheritance copies the local vocabulary and user-specified productions from the parent grammar to the inheriting grammar (36:5-20).

4.4.1 Domain Model and Grammar. The ToolKit domain model is composed of object classes defined as subtypes of objects contained in the core language domain model. There are three major object classes in the Mathematical ToolKit: symbols, predicates, and expressions. Using an incremental building approach, the extensive number of objects and their corresponding grammar productions were incorporated into the final ToolKit version. In terms of structure, the abstract syntax trees constructed by the ToolKit are

automatically appended to the core language tree whenever a ToolKit element is processed by the parser.

4.4.2 Validation. The validation phases of the ToolKit parser were essentially the same as the phases for the core language, except for two notable points. First, in the area of optimization, it was more difficult to remove ambiguities from the ToolKit grammar. The inheritance relationship increased the grammar's complexity and propagated previously nonexistent errors from the core language into the ToolKit, making resolution difficult and time-consuming. Ultimately, the final parser contained two unyielding reduce/reduce errors, which upon testing did not demonstrate any miscreant behavior. The second distinction was in the area of source program parsing. Since the coverage of the combined parsers was greatly expanded, the complexity of the input files was also increased.

4.5 Summary

This chapter detailed the design and implementation of a robust and fully operational *Z* language parser within the REFINE environment. This product permits the scanning of *Z* specifications into hierarchical tree structures which, in turn, facilitate a formal mapping between the abstract source language and the wide-spectrum target language. It is this mapping that forms the solid basis for the development of the next generation application system. The next phase builds on this foundation through the unification of the *Z* and LARCH (23) parsers into a composite REFINE target model, outlined in Chapter V.

V. *The Design and Implementation of a Unified Domain Model*[1]

5.1 *Introduction*

In Chapter I, two complementary paths, a set-theoretic, or model-based approach and an algebraic, or theory-based approach, were presented for formalizing object transformations into the REFINE object base. Figure 5.1 depicts this transformation process and also shows a third horizontal path. This path represents another transformation mechanism, one intended to produce a unified model of the designated formal specification languages, *Z* and LARCH.

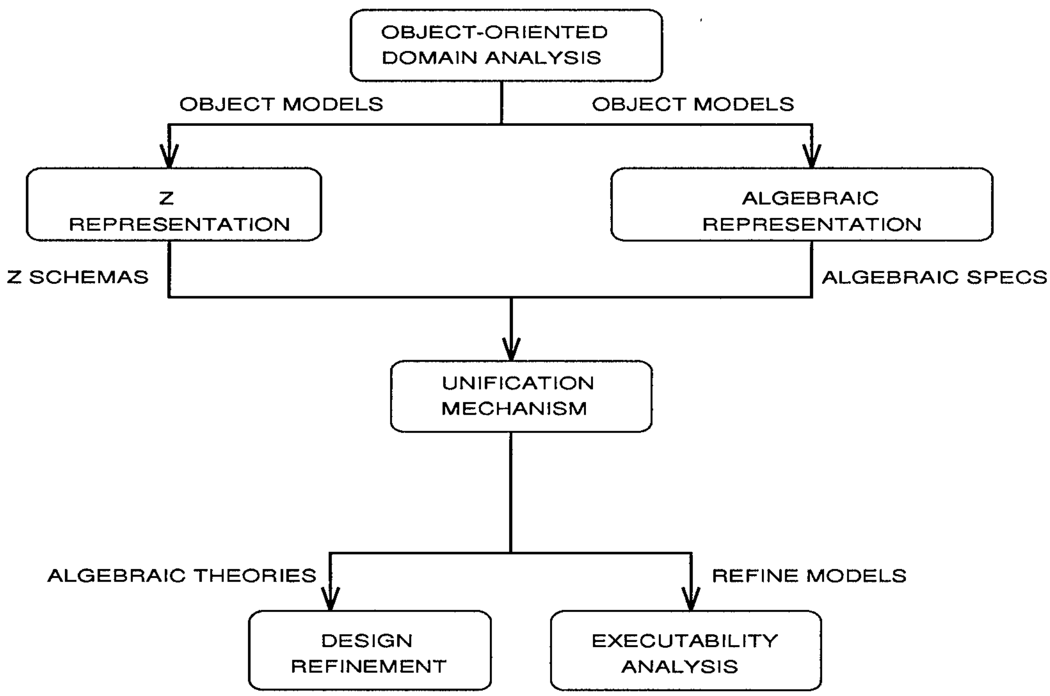

Figure 5.1 Formalized Object Transformations

In order to automate a portion of the first transformation process, both vertical paths contain a step for language-specific compilation. Providing an initial operational capability

---
[1]This chapter was co-written with Captain Catherine Lin. It also appears in (23).

toward this goal, robust language parsers were designed and implemented. These parsers generated similar structural representations (abstract syntax trees) that established a basis for the preliminary analysis and design of a unified target model.

This chapter focuses on the evolution of a Unified Domain Model, whose aim is the creation of a single, cohesive framework for translating different source specifications into a unifying abstract structure. Four iterations define this evolution, and they are detailed in Sections 5.2 through 5.5:

1. Evaluation of Abstract Syntax Trees (AST)
2. Analysis of Design Alternatives
3. Development of a Unified Core Model
4. Development of Language Specific Extensions

5.2 Evaluation of Abstract Syntax Trees

As mentioned in Section 5.1, the parser-generated ASTs provided a common focal point for analyzing the structures of LARCH and Z to establish the design of a unified target model. This evaluation process was likened to an electronic balanced mixer circuit. Accepting two input frequencies, a balanced mixer produces four outputs: the two original frequencies, the sum, and the difference. Considering the two ASTs as input, the evaluation produced the originals, common core objects, and language specific objects, as depicted in Figure 5.2.

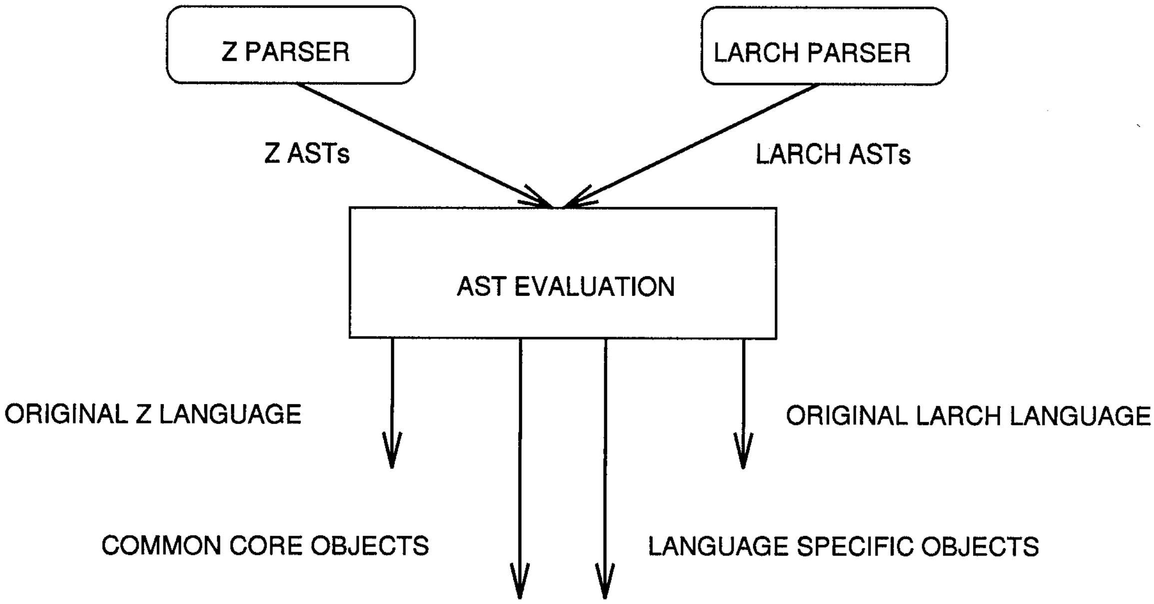

Figure 5.2 AST Evaluation Process

5.2.1 Common Core Objects. The evaluation of the ASTs revealed that both languages contain strong similarities in their conceptual representations of a specification. Both languages require a set of signatures, axioms, and external references to describe a problem domain. LARCH uses operators with signatures, axiomatic equations, and context references in its specification format. *Z* defines a specification using signatures, predicates, and schema referencing. Even though the syntactical domain, or notations, differ, the semantic domain, i.e., the mathematical foundation, remains fundamentally the same. Consequently, Table 5.1 identifies a set of core objects extracted from both languages that form the unified domain model.

5.2.2 Language Specific Objects. The AST analysis also revealed differences in the way each language characterized the common objects. For example, even though the ASTs identify that both languages contain a signature, a *Z* signature can use specific

Table 5.1 LARCH and *Z* Commonalities

Unifying Object	LARCH	*Z*
Theory Object	Trait	Schema Paragraph
Theory Id	Trait Id	Schema Id
Theory Body	Trait Body	Schema
Theory Signature	Introduces Clause	Declarations
Theory External Reference	Context References	Schema References
Theory Axioms	Asserts Clause	Axiom Part

notions of state transitions, both the *before state* (State) and the *after state* (State'), while LARCH does not. Additionally, the *Z* language is capable of explicitly declaring input and output variables, while LARCH cannot. Therefore, these variances required an approach that captured each language's specialization, yet preserved the core characteristics.

5.2.3 A Framework for Language Inheritance. One approach for modeling the common core objects and the specialized objects is through language inheritance. Language inheritance establishes a notion of a common base language that is inherited by various *dialects* (36). These dialects specialize the common language in order to implement language specific constructs. This approach seemed well-suited for modeling the specialized constructs of LARCH and *Z* because each language can be represented as a *dialect* of a common core language, i.e., a mathematically-based language composed of signatures, axioms, and external references. Establishing this framework in REFINE supports the development of a uniform interface for formalized object-oriented models. The REFINE object base is capable of representing inheritance and specialization through object classes and subclasses. Figure 5.3 illustrates an example of language inheritance for LARCH and *Z* using the unifying objects defined in Table 5.1.

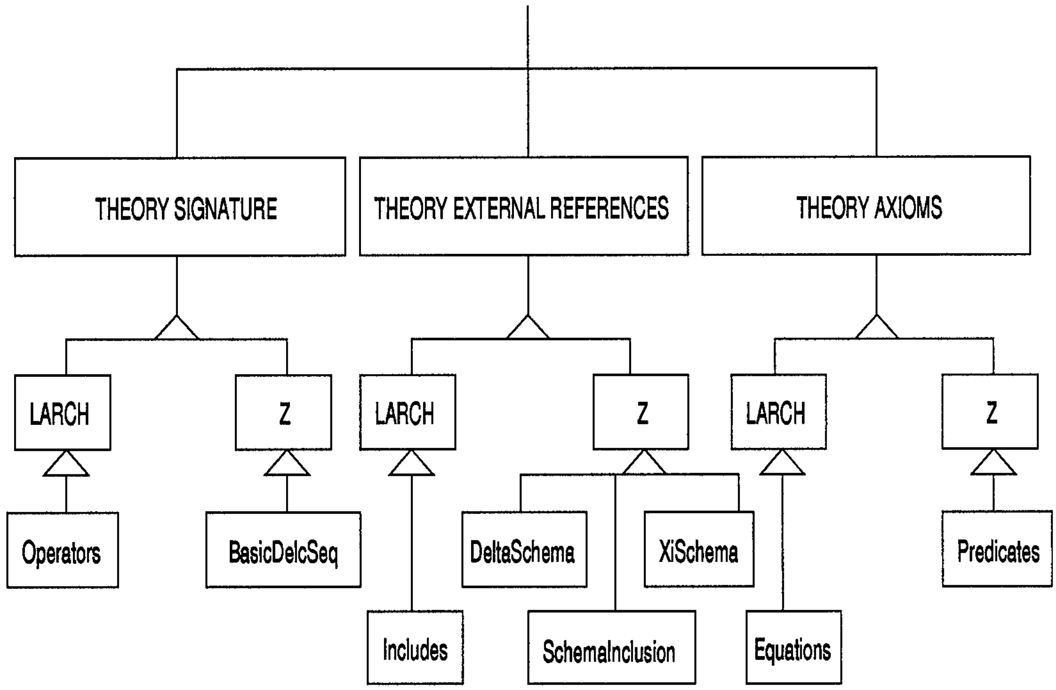

Figure 5.3 Language Inheritance

5.3 Analysis of Design Alternatives

Using language inheritance as a framework to consolidate the similarities and differences between LARCH and *Z* modified the initial transformation process depicted in Figure 5.1. Instead of converging to a unified model after compilation, the new design focused on unifying the two languages during the parsing stage. This new process is founded on the common base language established above and is shown in Figure 5.4.

In order to implement this new design, two possible courses of action were developed:

1. Develop transformation programs for each language to convert the grammars into the unified model.

2. Refine the original language grammars in order to parse into the unified domain model.

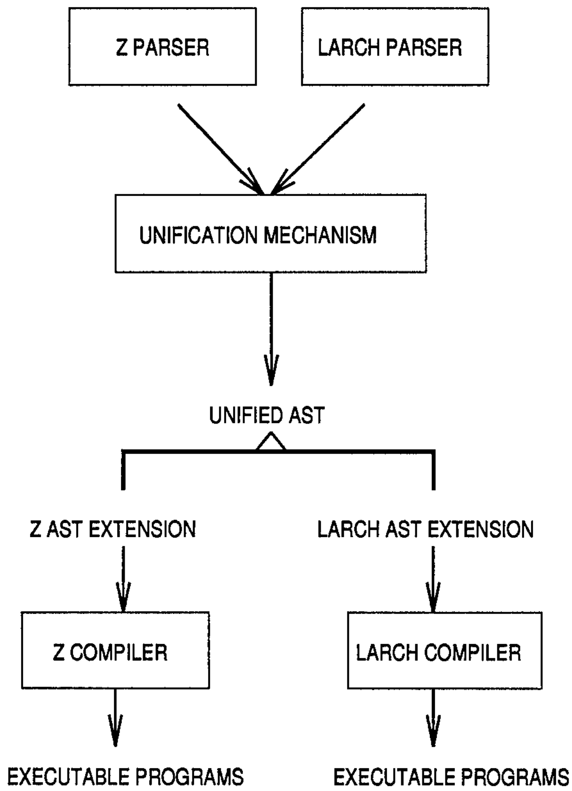

Figure 5.4 Modified Transformation Process

Since these paths were significantly different, an in-depth analysis was conducted to select the best candidate. The strengths and weaknesses of each alternative are summarized in Figure 5.5 and discussed in the two subsequent sections.

5.3.1 Transformation Approach. Even though the transformation approach decreases the parser's complexity, the process introduces a level of informality into the unified implementation. This informality results from relying on hand-coded algorithms to construct the unified model instead of a more established parsing technique that relies on machine-generated algorithms. Also, since reusable code is essentially prohibited, the transformation approach increases the amount of transform specializations needed in the

UNIFICATION APPROACH ALTERNATIVES

	ADVANTAGES	DISADVANTAGES
TRANSFORM APPROACH	LESS COMPLICATED COMPILERS MORE EXECUTION MODEL ANALYSIS	ONE MORE LEVEL OF TRANSLATION INCREASED TRANSFORMATION SPECIALIZATIONS
PARSING APPROACH	ONE LESS LEVEL OF TRANSLATION TRULY COMMON CORE POSSIBLE SET OF WELL-DEFINED REFINE ABSTRACTIONS	MAY COMPLICATE COMPILERS SACRIFICE WORK ON EXECUTION MODEL

Figure 5.5 Unification Design Alternatives

overall formal process. More importantly, despite the advantage of completing more analysis of the execution model, this model does not adhere to the desired correctness constraint.

5.3.2 Parsing Approach. The parsing approach complicates the development of the formal language compilers, thereby placing a time constraint on performing a complete execution model analysis. But, this approach preserves each language's syntactical structure in the unified implementation. By using DIALECT to accept both the unified domain model and each language's surface syntax, DIALECT generates a parser for Z and a parser for LARCH that map to the same common core model. Since this approach preserves each language's syntax, thereby fulfilling the correctness requirement, it is the strongest candidate for the unification implementation.

5.4 Development of a Unified Core Model

After completion of the AST evaluations and approach analyses, the next iteration in the evolution of the unified domain model focused on the common core objects. Based on previous experience with the original parsers and the sensitivity of the DIALECT tool, an incremental strategy was adopted to reduce the overall complexity of this development phase. Using the three OOA models as increments, the following steps were outlined to parse both LARCH and Z into the unified core:

1. Design a Unified Core Domain Model
2. Map Language Commonalities to Unified Core Syntax
3. Implement the Core Domain Models and Grammars
4. Compile and Validate Grammars

The following sections detail these steps and the resultant initial capabilities of the "ULARCH" and "UZed" parsers. The fully operational versions are described in Section 5.5.

5.4.1 Framework for the Unified Core Domain Model. The common object classes identified during the AST analysis in Section 5.2.1 formed the basis for the unified domain model. In order to formally relate these commonalities and their intrinsic properties, an encapsulating framework was defined. Comparing both LARCH and Z at a high level of abstraction, we can generally characterize these languages as theory presentations, or sets of statements used to describe some problem domain (4). Examples of theory statements include domain models, software systems, and formal specifications. Since both LARCH and

Z define a specification as their top-level object, a corresponding root object in the unified core was defined as a "domain theory". Mapped to the Rumbaugh OOA framework, this domain theory was categorized into three distinct DomainTheoryTypes: ObjectTheory, DynamicTheory, and FunctionalTheory. In terms of multiplicity, one object theory is required, while there may be zero or more dynamic and functional theories. To complete the domain theory framework, each theory type was defined through identical aggregate components: a TheoryId and a TheoryBody. Figure 5.6 provides a graphical depiction of the core framework.

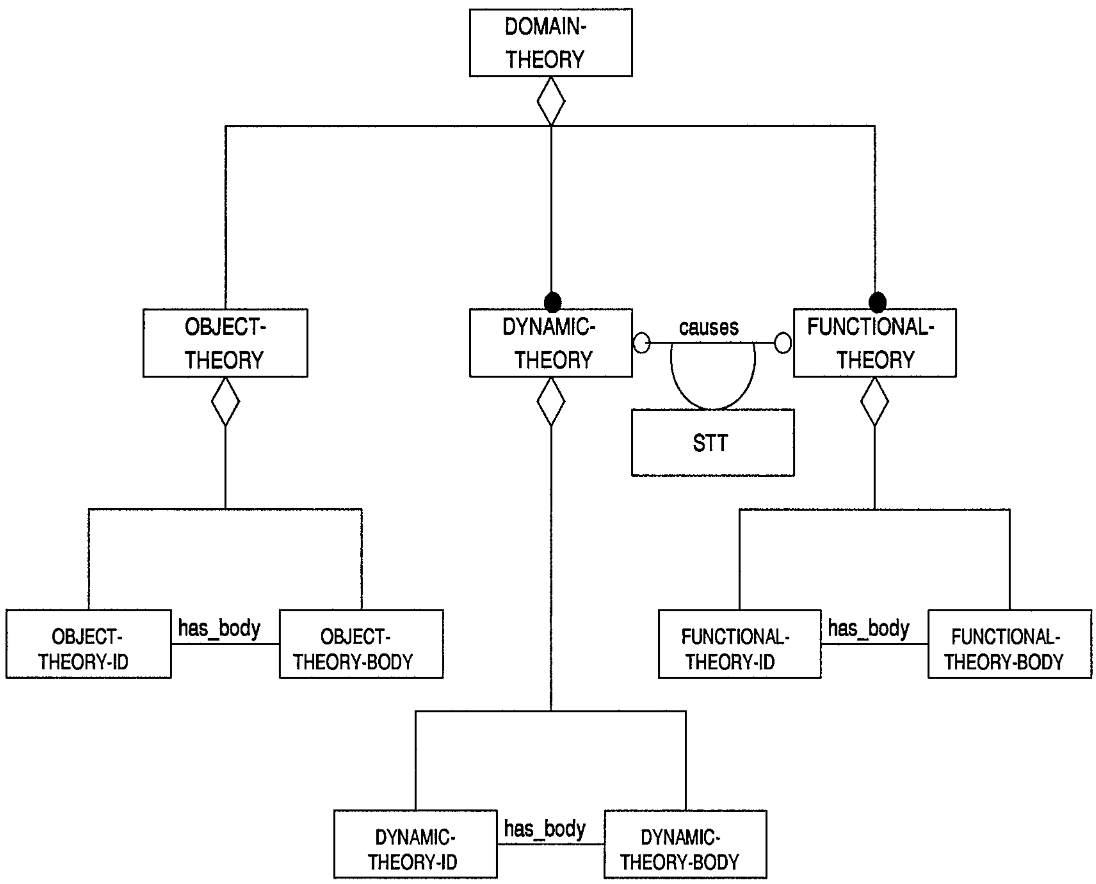

Figure 5.6 Unified Core Domain Model

As stated above, a complete cycle of design, development, and validation activities was accomplished for each of the domain theory types. Based on the fact that every domain theory must have an object theory, the first phase concentrated on the ObjectTheory implementation. The entire process is detailed in the following three subsections. Since the other two phases for the dynamic and functional theories mirrored the first phase, their development is summarized in Subsection 5.4.5.

5.4.2 ObjectTheory Mappings. After establishing the core domain framework, the ObjectTheory commonalities of LARCH and *Z* were implemented. Starting with the layer composed of the ObjectTheoryId and ObjectTheoryBody classes, additional common object classes were identified. Since the ObjectTheoryId is a leaf node, no further decomposition was possible. The ObjectTheoryBody, however, was partitioned into two additional layers. The first tier contained the ObjectTheoryDeclarations and ObjectTheoryAxioms. The second tier, containing SignatureDeclarations and ExternalReferences, resulted from further decomposition of the theory declaration component. Figure 5.7 illustrates all the common layers within the ObjectTheory class. Below these layers, the composition of the two source languages became too divergent, thereby requiring language specific extensions. These extensions are detailed in Section 5.5.

5.4.3 Implementation of Domain Models and Grammars. The establishment of a stable domain model for the object theory facilitated the development of the corresponding REFINE code. The object classes and associations in the domain model mapped directly into REFINE'S object classes and map attributes. Analogous to the development of the original LARCH and *Z* parsers, the tree-attribute structure was defined by using object

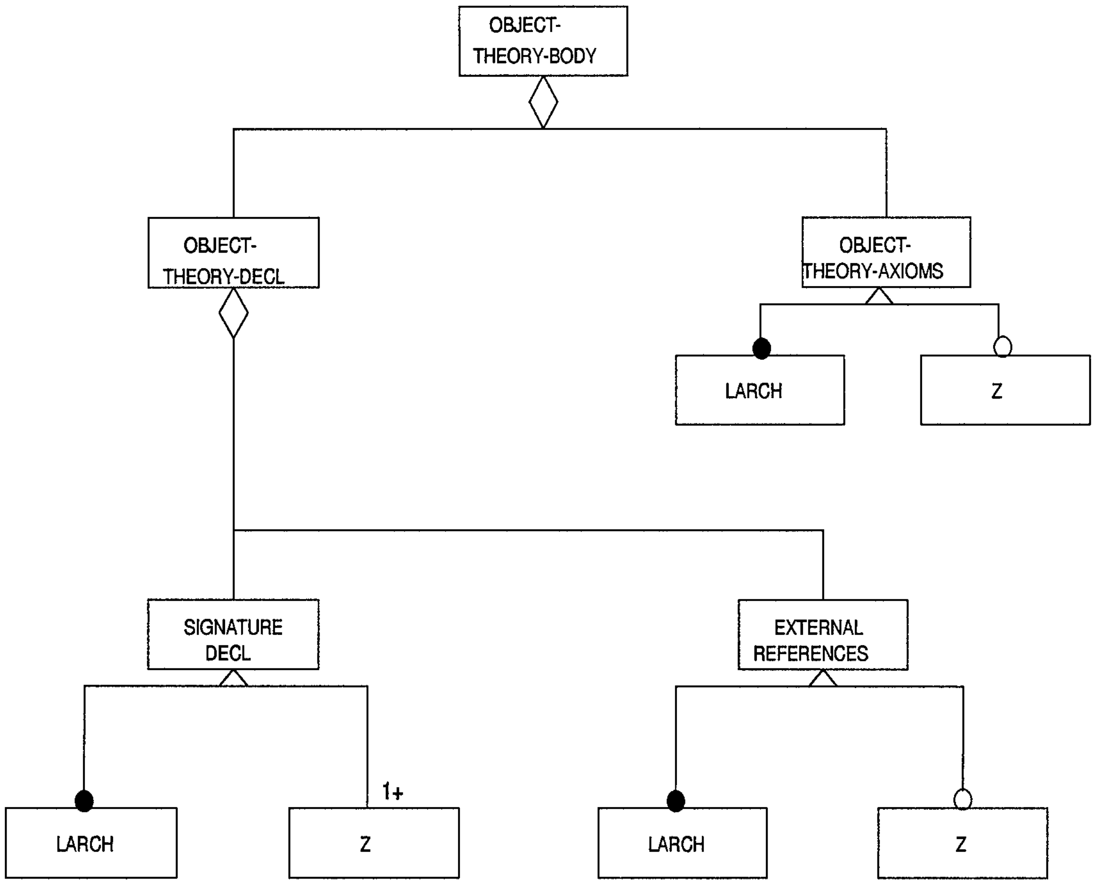

Figure 5.7 Object Theory Body Domain Model

classes for the nodes and the corresponding map attributes for the links. This tree-attribute hierarchy guided the modifications for each language's grammar. Maintaining consistency, LARCH and Z specific rules were carefully mapped into a target unified object theory rule. In addition to re-naming terminals and non-terminals to reflect the common unified objects, the production rules had to correctly map to each formal language's extensions.

5.4.4 Compilation and Validation of Grammars. As a direct result of each original grammar's rigorous development, the compilation of the unified grammars was straightforward. The majority of errors resulted from ill-defined attributes connecting the

unified model to the language specific models. The validation process for the unified parsers was accomplished in two phases: parsing of source programs and AST evaluations. Since this iteration of the unified parsers was focused on the object theory, just the object models of the counter and fuel tank domains introduced in Chapter III were used as inputs. A visual inspection of each domain's object theory clearly identified the boundary between the common object classes and the specialized ones. Traversing the two ASTs, each node was inspected to verify the presence of expected attribute values. As an additional confirmation, the ULARCH and U*Z*ed ASTs were compared against their respective language-specific ASTs. This process verified the preservation of each language's syntax and semantics, thus validating a common framework for parsing *Z* and LARCH specifications.

5.4.5 DynamicTheory and FunctionalTheory Iterations. As stated in Subsection 5.4.1, the iterations for both the dynamic and functional theories were symmetrical to the object theory iteration. The object classes of TheoryDeclarations, SignatureDeclarations, ExternalReferences, and TheoryAxioms were all instantiated with instances of DynamicTheory and FunctionalTheory. Complete graphical models are located in Appendix D, while Appendices E and F contain the corresponding REFINE implementations. For the validation of both theory types, the counter and fuel tank domains were augmented with the appropriate dynamic and functional models. The confirmation of this stage baselined the Unified Core Model's capabilities. In order to complete the functionality of these two unified parsers, the final phase required the addition of language specific extensions.

5.5 Development of Language Specific Extensions

Since the top-level hierarchy of both languages inherited from the common core framework, the required changes for the language extensions consisted of mapping each specific syntax to the inherited core. Reusing the language specific objects and map attributes defined in the original parsers, the only modifications consisted of re-defining the map attributes that linked the core framework with the language specific extensions. As an example, Figures 5.8 and 5.9 illustrate the ULARCH and UZed specific extensions of the inherited SignatureDeclaration class.

Although the majority of changes for both language extensions was similar, some individual syntax alterations were also necessary. Specifically, for ULARCH, LaTeX notations were required in the grammar. Since both LARCH and *Z* specifications use LaTeX notations, the production rules of the ULARCH grammar were changed to incorporate the LaTeX notation as well. Also, since there are significantly more syntactical units in *Z* than in LARCH, the UZed extension process was compounded. The following list details the major tasks involved:

1. Distinction Between State and Event Schemas - Correlated to the original parser, UZed's DynamicTheory object was further partitioned into StateTheory and EventTheory subtypes. Their internal structures were identical to their parent object, composed of declarations, external references, and axioms.

2. Inclusion of Definition Paragraphs - As detailed in Section 4.3.2, the original *Z* parser recognizes two major types of paragraphs: schema and definition. The schema paragraph types mapped directly to the unified object, dynamic, and functional theories

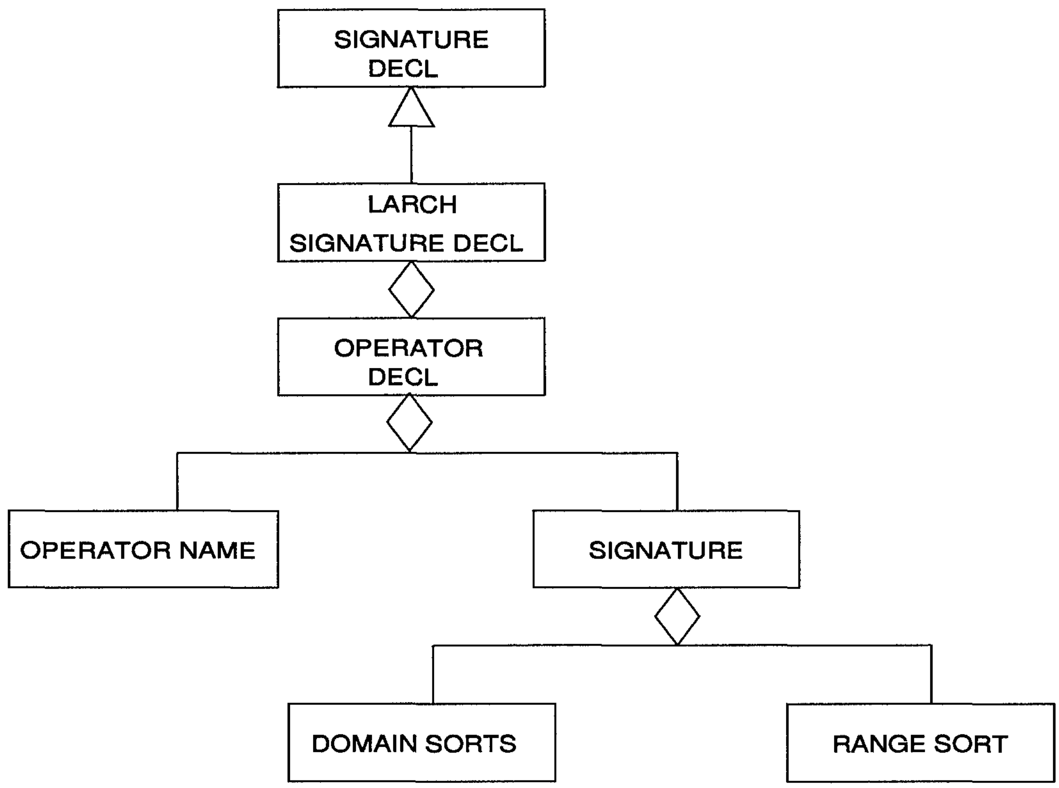

Figure 5.8 Signature Declaration - ULARCH Specific Extension

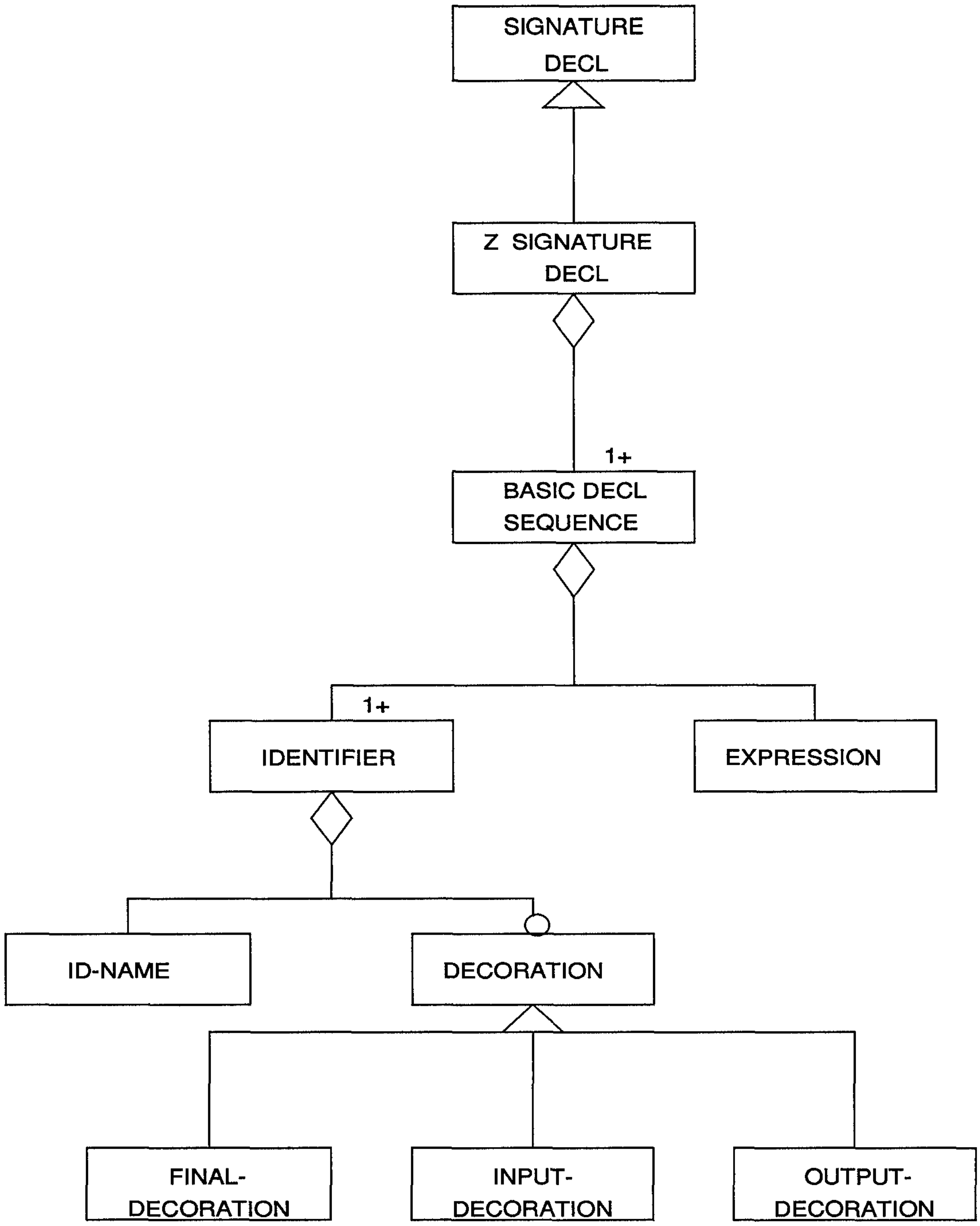

Figure 5.9 Signature Declaration - UZed Specific Extension

discussed above. The definition paragraphs were designated as another subtype of DomainTheoryType, namely a DefinitionTheory. Consequently, all subtypes of DefinitionTheory were successfully appended to the core framework.

3. Inheritance of the Mathematical ToolKit - In order to create a unified version of the ToolKit, the information pertaining to the inherited grammar was modified. Since the ToolKit is appended to the core *Z* parser at a low level, the changes to the UToolKit were minimal.

The complete set of diagrams for the U*Z*ed extensions are contained in Appendix G, while the REFINE code for the domain model and grammar are located in Appendices H and I, respectively. The equivalent ULARCH appendices can be found in Lin's thesis (23).

5.6 Summary

Although the syntactic domains of the two source languages, *Z* and LARCH, were different, their semantic domains were conceptually the same. Both formal specifications encapsulate theories about a target problem domain through sets of signatures and axioms. The notion of language inheritance produced a framework for a unified core domain model and specialized language *dialects.* Intuitively, the languages' specializations should be minimal, since they are both founded on mathematical constructs; however, during this research effort, the lack of a *verifiable* mechanism for pattern matching and term rewriting limited the depth of the inherited core to a much higher level. Nonetheless, the resultant unified design does provide an initial consolidation of various formal language representations, which, in turn, promotes reusability and prevents escalation in the number of divergent execution models. Adhering to the predefined goals of completeness, consis-

tency, and correctness resulted in a stable unified domain model that formed the basis for a target execution framework, detailed in the next chapter.

VI. *Design of a Formal Execution Framework*[1]

6.1 Introduction

In Chapter V, Figure 5.4 depicted a transformation process that produces an executable program from a unified abstract structure. To maintain consistency, executable code generated from a formal specification requires a concrete target model. An execution framework is the initial step toward building this target model because it identifies the executable program's data structures and functions. The existing ULARCH and U*Z*ed parsers provide the syntactical basis for this execution framework, but additional semantic analysis is required to ensure consistency. Therefore, this chapter elaborates the steps necessary to complete the compilation analysis phase (refer to Figure 4.1) and then describes the design and development of an execution framework. The specific tasks are outlined below:

1. Semantic Analysis
2. Creation of an Execution Domain Model
3. Development of Execution Framework Mappings
4. Prototyping an Initial Executable Program

6.2 Semantic Analysis

The semantic analysis portion of the compilation process consists of various checks that ensure a program fits together in a meaningful way prior to generating an executable program (1:5). For this research effort, two semantic analysis tasks were identified to simplify the translation phase: shorthand expansion and type checking. In general, shorthand

[1]This chapter was co-written with Captain Catherine Lin. It also appears in (23).

expansion augments the AST by including the signatures and axioms of any referenced components. Type checking ensures that operators used in expressions are not applied to incompatible operands. Figure 6.1 illustrates the sequencing of these two tasks.

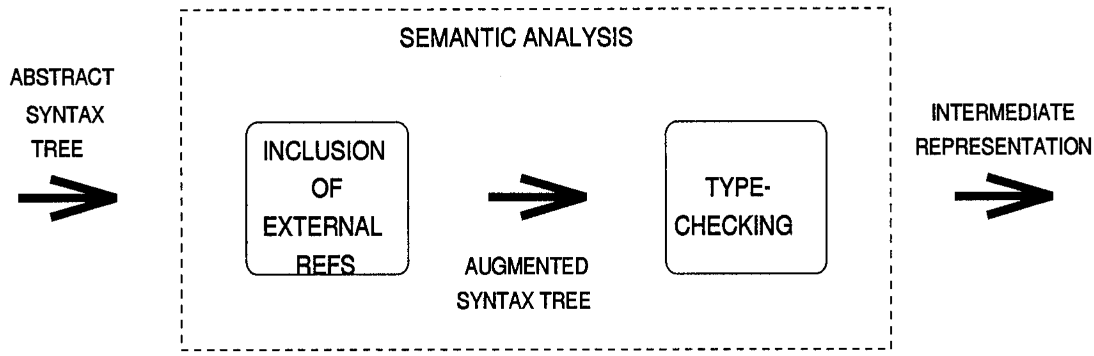

Figure 6.1 Semantic Analysis Phases

While the following subsections describe the analysis and design performed for the languages' shorthand expansion, it was decided that type checking would not be addressed during this research effort for two reasons:

1. The existence and accessibility of stand-alone type-checkers, i.e., MIT's Larch Shared Language (LSL) Checker (16) and DePaul University's *Z* Type Checker (ZTC) (18), provided an interim verification capability.
2. Since type checking is essentially an implementation task, it could be reserved for future enhancements.

6.2.1 Shorthand Expansion Analysis. In an abstract specification, shorthand notations are used to promote modularity and encapsulation through a reduction in the amount of formal text. To translate a specification into a concrete implementation requires the expansion of these notations to produce a robust intermediate representation. In a

LARCH specification, shorthand expansion consists of unioning the operators and axioms of the "includes" trait with the original referencing trait. In a Z document, there can be many different types of shorthand notation, e.g., the Δ/Ξ operations and the numerous schema calculus expressions, such as schema inclusion.

In terms of the unified model, the schema inclusion notation is comparable to LARCH's "includes" mechanism, whereby the schemas' signatures are merged and the predicates are conjoined (22:42). In contrast, however, the Δ and Ξ notations do not have LARCH counterparts. Described in Section 4.3.3, these prefixes can be attached to a schema name in order to indicate types of schema importation. To demonstrate the underlying semantics of the two conventions, consider the following schema, S, as a reference point.

$$\begin{array}{|l}
\multicolumn{1}{l}{S} \\ \hline
i, j : \mathcal{N} \\ \hline
i > j \\ \hline
\end{array}$$

Figure 6.2 Example Schema S

The succinct notation of the Δ and Ξ conventions are used to distinguish updates from observations. Specifically, Z's Δ convention, as in other areas of mathematics, is used to signify a change in state. Recalling that the tick (') decoration indicates a post-operation value, the implicit definition of this notation is given as: $\Delta State \mathrel{\widehat=} [State;\ State']$. The syntax of the expanded Δ schema for schema S is provided in Figure 6.3.

In Z, the Ξ symbol, chosen because of its similarity in appearance to the equivalence symbol ($\equiv$), signifies the state of the schema before and after an operation where all of the schema's components remain unchanged (22:45). The default definition of the Ξ convention

is as follows: $\Xi State \mathrel{\widehat{=}} [\Delta State \mid \mathrm{v1'} = \mathrm{v1} \land \mathrm{v2'} = \mathrm{v2} \land \ldots\ \mathrm{vk'} = \mathrm{vk}]$, where v1. .vk inclusive are the variables of *State*. A visual representation of an expanded Ξ schema is depicted in Figure 6.4.

$$
\begin{array}{|l}
\Delta S \\
\hline
i, j : \mathcal{N} \\
i', j' : \mathcal{N} \\
\hline
i > j \\
i' > j' \\
\hline
\end{array}
$$

Figure 6.3 Expanded Delta Schema

$$
\begin{array}{|l}
\Xi S \\
\hline
i, j : \mathcal{N} \\
i', j' : \mathcal{N} \\
\hline
i > j \\
i' > j' \\
i = i' \\
j = j' \\
\hline
\end{array}
$$

Figure 6.4 Expanded Xi Schema

To address the wide variety of shorthand notations, two alternative expansion approaches were evaluated. The first alternative combines the parsing task with the shorthand expansion, while the second alternative maintains a separation between syntax and semantics.

6.2.1.1 Addition of Explicit Production Semantics. Although the compilation analysis phase depicted in Figure 4.1 distinguished between the parsing and semantic analysis phases, some language compilers do merge these two tasks. Supporting this

approach, DIALECT possesses the capability to embed semantic analysis tasks within a language's production rules. The DIALECT Users Manual outlines two situations where these explicit production semantics are effective (36:5-17):

- Productions where the user must set attributes of an object that are not fully described by the syntax.
- Productions that are used to augment the syntactic inheritance hierarchy.

The second instance appeared to be applicable to this research effort; however, a prototyping effort yielded disappointing results. The production semantics, as defined in the DIALECT manual, did not appear to augment the ASTs, and it could not be discerned whether the added production semantics had any effect on the languages' rules. Furthermore, it seemed that the production semantics required terminal objects (i.e., the lowest level objects in the domain models) in its definition, which constrained the effective range of this approach. Therefore, this alternative was deemed inappropriate.

6.2.1.2 Development of Traversal Algorithms. The second alternative for augmenting the ASTs required the creation of tree traversal algorithms. Essentially, this methodology uses REFINE programs to manipulate the object base created during parsing, thereby expanding the shorthand notations. As a prototype, the following generic traversal steps were developed:

1. Locate the shorthand notation objects.
2. For each included object, collect its signatures and axioms.
3. Augment the object base with the collected signatures and axioms.

After prototyping the predefined algorithm, the resultant ASTs were viewed for accuracy and completeness. The test specification had been augmented with the collected signatures and invariants. Also, because this approach maintained disjoint syntactical and semantic phases, the grammars' production rules were not corrupted. Supported by this evidence, this alternative was selected for implementing shorthand expansions.

6.2.2 Implementation of Traversal Algorithms. For manageability, the implementation of the traversal algorithms was divided into two separate phases. Contained in Appendix J, the completed routines are depicted in the structure chart in Figure 6.5.

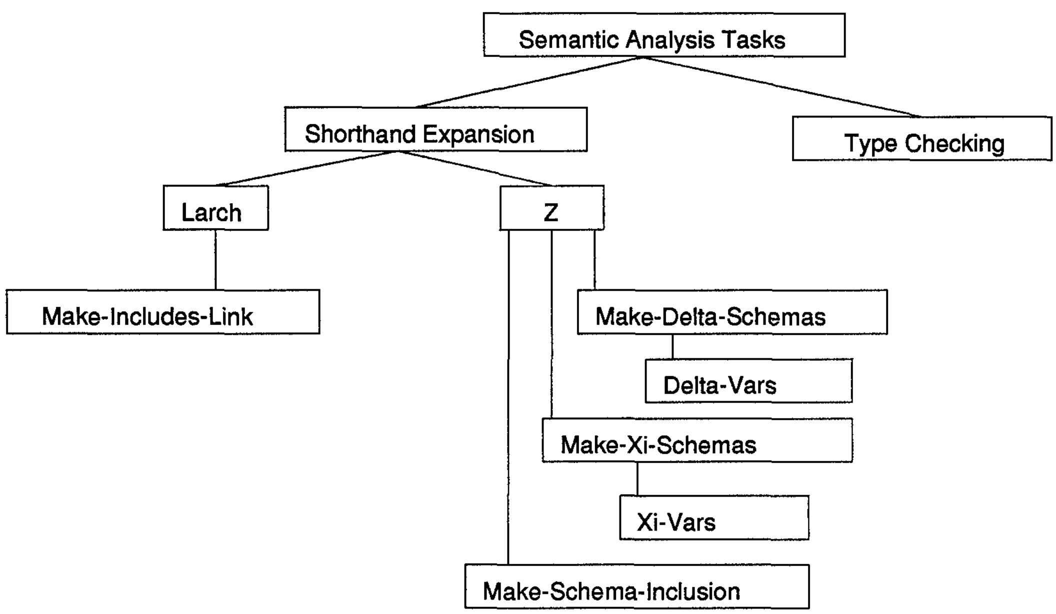

Figure 6.5 Traversal Routines

In the first implementation phase, the successful test algorithm was directly translated into routines for the ULARCH "includes" notation and U*Z*ed's schema inheritance. A summary of the respective routines is provided:

1. MAKE-INCLUDES-LINK: An "Including" trait's theory is expanded by unioning the *Included* trait's operators and axioms.

2. MAKE-SCHEMA-INCLUSION: A state schema's signature is unioned with the referenced object schema's signature. Additionally, the predicates of both schemas are conjuncted together within the state schema's predicate.

For the second phase of the iteration, based on frequency of use and correlation to the Rumbaugh model, the more complicated Δ and Ξ notations were implemented. Due to time constraints, the algorithms for the less frequently used Z schema calculus expressions were reserved for future enhancements. The specific routines developed for the two conventions are listed below:

1. MAKE-DELTA-SCHEMAS: A Δ reference is expanded by unioning the signature of the Δ schema with the signature of the referenced schema. Likewise, the predicates of both schemas are conjuncted together. A boolean annotation attribute, *delta-link*, is set to true.

   - MAKE-DELTA-VARS: If its *delta-link* attribute is true, the Δ signature is expanded a second time to include the ticked counterparts of the referenced object's variables. Also, the predicate is expanded to include the corresponding constraints placed on those ticked variables (See Figure 6.3).

2. MAKE-XI-SCHEMAS: A Ξ reference is expanded by unioning the signature of the Ξ schema with the signature of the referenced schema. Likewise, the predicates of both schemas are conjuncted together. A boolean annotation attribute, *xi-link*, is set to true.

- MAKE-XI-VARS: If its *xi-link* attribute is true, the Ξ signature is expanded a second time to include the ticked counterparts of the referenced object's variables. Also, the predicate is expanded to include the corresponding constraints placed on both the ticked and unticked variables (See Figure 6.4).

6.2.3 Validation of Algorithms. The testing and validation of the traversal algorithms consisted of visually inspecting the augmented ASTs with OBJECT INSPECTOR. During this process, two types of errors were detected: misplacement within the hierarchy and replication of variables. In the first case, incorrect attribute references caused the "included" signatures and axioms to be appended to the wrong parent class. In the second case, the temporary set used to collect the referenced signatures and axioms was not being emptied between enumerations. Upon correction of these errors, the traversal algorithms generated the correct results, thereby producing robust intermediate representations of the source languages. The next phase of development focused on the target execution framework.

6.3 Creation of an Execution Domain Model

After completing the compilation analysis phase, the first step in designing an execution framework required the creation of a domain model for the target execution language, REFINE. Analyzing the syntax of the language produced four major object classes as the target data structures: object classes, map attributes, functions, and rules. The object classes and map attributes were identified as terminal objects, while the function and rule classes each decomposed into local parameters and a set of preconditions and postcon-

ditions. Figure 6.6 illustrates the major object classes of the execution domain model. Once defined, the objects and associations in the domain model mapped directly into the object classes and map attributes of a REFINE description. This description reflected the execution domain model, thus establishing a concrete target for the execution framework mappings. (See Appendix K for the complete graphical model and Appendix L for the corresponding REFINE implementation.)

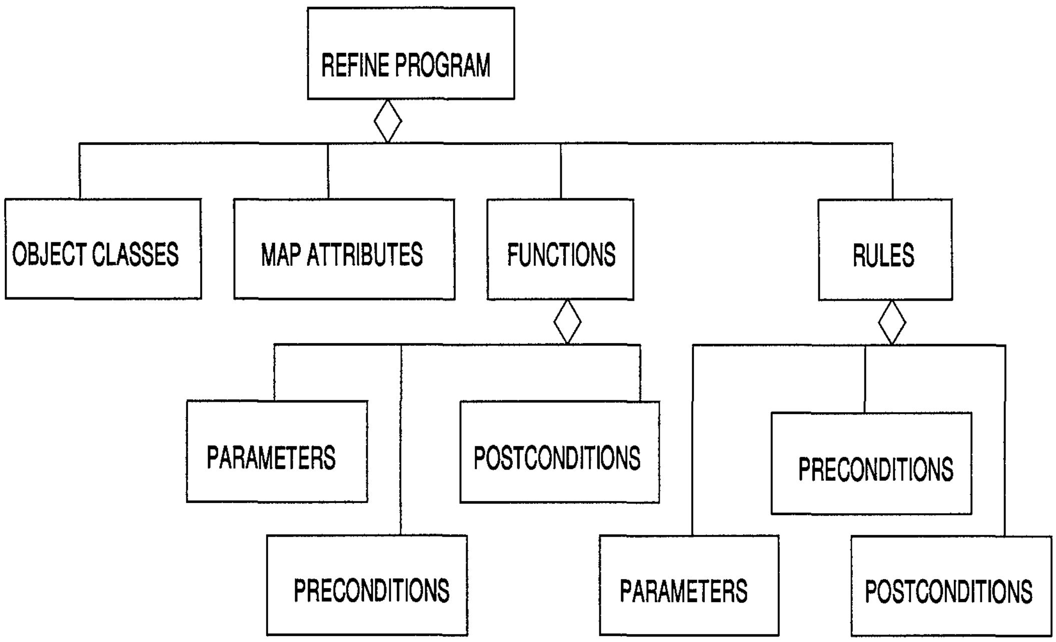

Figure 6.6 Execution Target Domain Model

6.4 Development of Execution Framework Mappings

Developing a translation process between the source and target domain models required mapping the ULARCH and UZed objects to the corresponding REFINE objects. Three execution maps were developed to correlate with the unified framework: an object theory, a dynamic theory, and a functional theory map. A two phase process was used for

each mapping. First, an initial translation model defined the appropriate REFINE object classes. Since an object theory models the structure of a system and may identify some initialization routines, the target objects for this mapping were defined as REFINE object classes, map attributes, and functions. REFINE functions were also defined for both the dynamic and functional theories, since the former captures a system's static states and events, while the latter models data transformations.

The second phase of each mapping process required an analysis of both the ULARCH and UZed models in order to match their individual structures to the corresponding REFINE target components. While similar dynamic and functional theory mappings were produced, the object theory mappings were significantly different because UZed's object theory predicates are realized after state initialization, i.e., within the subsequent dynamic and functional theories. As a result of this difference, the lower right portion of Table 6.1 remains empty, while Table 6.2 is complete.

Table 6.1 ULARCH and UZed Object Theory Maps

Execution Target Object	ULARCH Source Object	UZed Source Object
Object Class Name	Theory Id	Object Theory Id
Map Attributes	Tuple Object	Object Theory Declarations
Attribute Name	Tuple Field Ids	Basic Identifiers
Attribute Domain	Theory Id	Object Theory Id
Attribute Range	Tuple Field Sort Id	Any Expressions
Functions	Operators	
Function Name	Operator Name	
Function Parameters	Quantifier Variable Ids	
Function Parameter Type	Quantifier Sort Id	
Function Return Type	Operator Range Id	
Function Let Statements	Theory Axioms	
Function PreConditions	Theory Axioms	
Function PostConditions	Theory Axioms	

Table 6.2 ULARCH and UZed Dynamic and Functional Theory Maps

Execution Target Object	ULARCH Source Object	UZed Source Object
Functions	Operators	State/Event/Fun Theory Body
Function Name	Operator Name	State/Event/Fun Id
Function Parameters	Quantifier Variable Ids	State/Event/Fun Declarations
Function Parameter Type	Quantifier Sort Id	Any Expressions
Function Return Type	Operator Range Id	
Function Let Statements	Theory Axioms	Identifiers
Function PreConditions	Theory Axioms	Any State/Event/Fun Predicates
Function PostConditions	Theory Axioms	State/Fun Post Predicates

6.5 Prototyping an Initial Executable Program

The feasibility of generating code from the execution maps was demonstrated by prototyping an executable shell for the counter and fuel tank specifications. The execution shells contained the required object classes, map attributes, and function names for creating an executable program. The shells were created by a sequence of four translation algorithms, illustrated in Figure 6.7 and described below:

1. *Make Object Class:* Translates application objects in a formal specification into REFINE object classes.

2. *Make Attributes:* Finds the formal representation of an object's attributes and translates them into map attributes with the appropriate domain and range values.

3. *Make Functions:* Translates the formal operations defined in ULARCH and UZed and creates a corresponding function name.

4. *Print Program:* Outputs the executable shell. The appropriate object classes, map attributes, and function names are listed in the correct REFINE format.

Appendix M contains the initial translation algorithms.

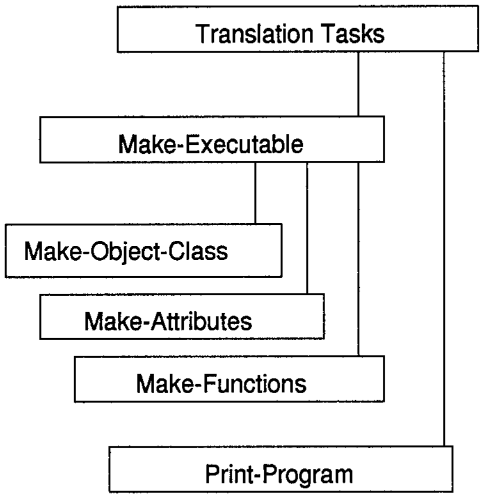

Figure 6.7 Translation Algorithms

6.5.1 Deficiencies in the Execution Maps. Constrained by time, the execution mappings were limited to a subset of the major object classes within the REFINE target domain model (refer to Figure 6.6). As a result, the initial prototyping effort did not complete the translation mechanism for the target model's function and rule object classes. These uncorrected deficiencies are detailed below.

1. *Make-Functions:* This mapping does not currently translate formal parameters and theory axioms into appropriate function parameters, preconditions, and postconditions. The translation must differentiate between theory axioms used as preconditions and those used as postconditions.

2. *Make-Rules:* The algorithm and mapping for this REFINE object class does not exist. This translation determines the sequences of events based on the state transition tables.

6.6 *Summary*

Developing an execution framework proved extremely useful for demonstrating the feasibility of generating code from a formal specification. The framework established the target data structures and functions for a consistent and correct translation process. It also became the foundation for developing the execution maps from the unified source models into the REFINE target model. From the execution maps, specific translation algorithms were created to generate executable REFINE code for the ULARCH and U*Z*ed counter and fuel tank specifications.

VII. Conclusions and Recommendations

This chapter provides a summary of the accomplishments of this thesis effort. Additionally, it describes the conclusions that can be drawn from this work and details some recommendations for further research in this area.

7.1 Summary of Accomplishments

The objective of this research was to establish a foundation for a next generation application composition system via the development of a formalized object-based transformation process. The specific goal was stated in Chapter I:

> *Formalize the object-based composition approach via the transformation of Z-based domain models into an executable framework, while incorporating a unification with corresponding algebraic-based models.*

Commencing with an extensive literature review of six topics relevant to the design of a unifying *Z* compiler, the resultant knowledge base outlined the basic disciplines for the product's development, i.e., flexibility, extensibility, reliability, and verifiability. To that end, an evolutionary development strategy was used to partition the compiler's development into three major iterations:

1. *Z* parser development (including Mathematical ToolKit)
2. Unified model development
3. Execution framework development

Implemented within predefined constraints of coverage, consistency, and correctness, these major evolutions resulted in a robust and accurate framework for the formal trans-

formation of object-oriented domain models, thereby supporting the feasibility of a next generation application composition system.

7.2 General Conclusions[1]

The following general conclusions can be drawn from this research:

1. Formal languages contain a basic set of core constructs. Even though formal languages' syntactic domains differ, they are all fundamentally based on mathematics. This common core identifies a canonical representation that can be inherited by the various language notations, or *dialects.* This canonical form is essentially a fundamental statement to which other statements can be reduced without a loss of generality (30). Analysis of the abstract syntax trees (ASTs) for LARCH and *Z* identified these fundamental constructs and they, in turn, evolved into the unified core domain model. The notion of language inheritance fulfills the *essence of reuse* by creating a consolidated framework for formal languages (3).

   Additionally, this consolidated framework establishes a front-end for formal system composition. This front-end can support three principal processes: design refinement, theorem proving, and code generation, as depicted in Figure 7.1. The unified framework can produce synthesis diagrams to support the notion of theory-based design refinement (44) (6). Theorem proving sentences can also be generated from the unified model for input into a theorem prover to verify a specification. Finally,

[1]This section was co-written with Captain Catherine Lin. It also appears in (23).

the unified model forms the basis for creating interface languages for the purpose of specification execution.

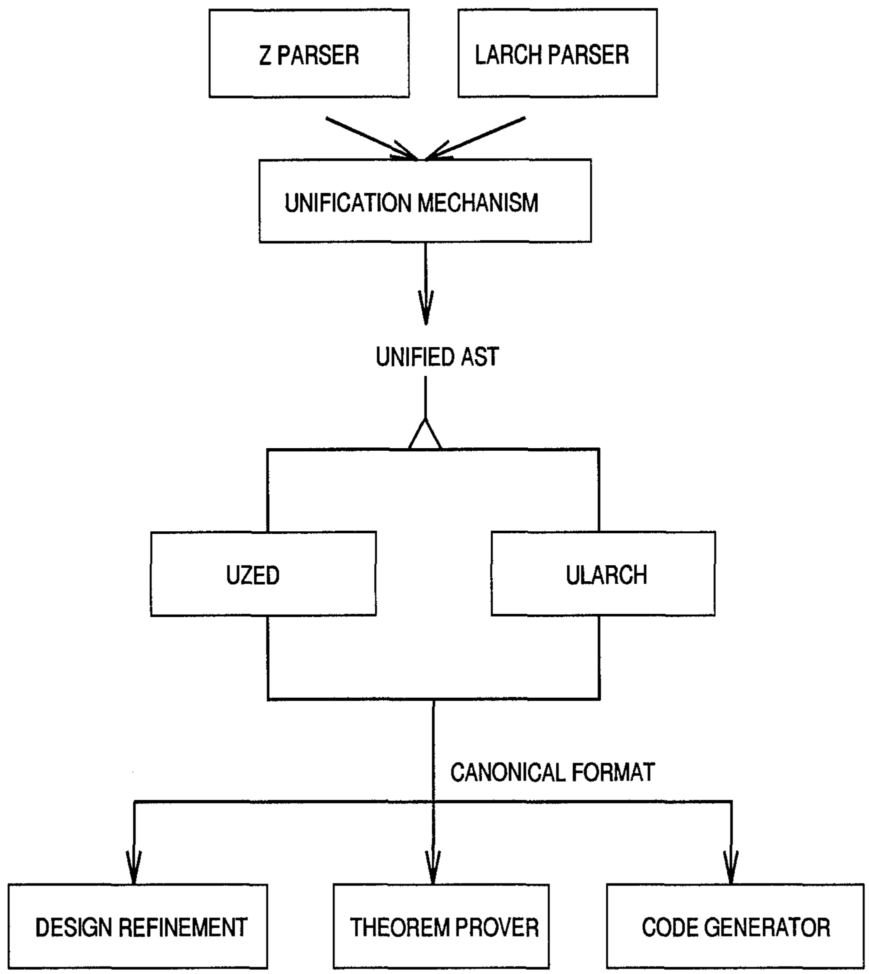

Figure 7.1 Expanded Transformation Process

2. The evolutionary development strategy is well-suited for developing a formal language compiler because the compilation process can be decomposed into several stages. Each stage produces an operational product demonstrating a minimum set of capabilities, and each incremental product is easily extensible.

3. Developing object-oriented domain models for LARCH and *Z* clarified the languages' structures by identifying the important objects and associations. The established domain models guided the development of the grammars, thereby constructing each language's syntactic hierarchy.

4. The REFINE environment simplifies the development of a formal language parser and an initial execution framework by providing compilation tools, i.e., DIALECT and OBJECT INSPECTOR. DIALECT handles token formation and creates an explicit hierarchy for *Z* using a BNF notation. These hierarchies, in the form of ASTs, are stored in the object base and viewed using OBJECT INSPECTOR. A visual representation is extremely beneficial because it allows one to inspect the parse tree and understand the underlying structure of the formal language. ASTs also provide a strong foundation to perform semantic analysis and begin building an execution framework.

7.3 Specific Conclusions

The following specific conclusions can be drawn about the *Z* and U*Z*ed parsers and the formal transformation of object-based specifications.

1. A unified parser for the *Z* and the LARCH specification languages successfully demonstrates the *feasibility* of using *Z* as a theory-based specification approach. The underlying canonical form of the composite framework promotes reusability and a single focal point for the execution model.

2. The grammar inheritance mechanism of DIALECT preserved the *de facto* relationship between the core *Z* language and the Mathematical ToolKit. This feature not only

supports the generally accepted criteria of encapsulation and modularity, but it also bolsters the acceptance of this tool within the *Z* community.

3. The *Z* and U*Z*ed parsers provide extensive coverage of the language's syntax (based on the First Edition of Spivey's notation (45)). Their robustness was demonstrated by the successful handling of a wide variety of specification documents. (See Appendices A through C for explicit examples.)

4. The *Z* and U*Z*ed parsers generate well-defined abstract syntax trees that simplify the semantic analysis and code generation phases of the compilation process. Specifically, for this research effort, the ASTs simplified the semantic analysis tasks of shorthand expansion and the initial execution translation described in Chapter VI. Additionally, the AST structures provide the capability for constructing an interface to other language-based tools, such as theorem provers.

5. The *Z* and U*Z*ed parsers are extensible. *Z*'s domain model and syntax allow developers to extend the current language model in a straightforward fashion by defining new subclasses. This flexibility will prove useful when the parsers are updated to the Second Edition of Spivey's notation (46).

6. REFINE's direct support of set theory and set-based operations was expressly beneficial to this research effort. Specifically, the "enumerate" construct of the REFINE language was well-suited to the implementation of *Z*'s semantic analysis tasks, i.e., shorthand expansion.

7.4 *Recommendations for Further Research*

The following issues should be addressed in future research efforts:

1. *Further Validate the UZed Parser* - A preliminary collection of specification test cases was used to complete the initial validation of the UZed parser. This established test set should be augmented with additional examples in order to further validate the coverage and consistency of the parser.

2. *Optimize the UZed Grammar* - The current implementation of the *Z* parser requires additional parentheses within certain predicates and expressions to reduce ambiguity. In future versions of the grammar, these constraints should be removed without corrupting the grammar or introducing spurious ambiguities.

3. *Extend the Unified Abstract Framework* - The goal for establishing a unified abstract framework is to provide a common base language with small variant specializations or *dialects*. However, the framework produced during this research does not reflect a complete common core because each language has not been reduced down to its most significant form. Decreasing the size of the language-specific extensions requires *pushing* the specializations down to the lowest possible level in the unified model. *Pushing* is defined as the consolidation and resolution of similar constructs via pattern matching and term rewriting routines. The pattern matching tasks are applied to the syntactic components of the ASTs and identify language commonalities. Mathematically verifiable procedures known as term rewriting are then used to translate the commonalities into a target canonical form.

4. *Implement a Comprehensive Suite of Semantic Analysis Tasks* - In addition to pattern matching and term rewriting, the following semantic analysis tasks are required for complete *Z* compilation:

   - *Expansion of Schema Calculus Expressions* - As discussed in Section 6.2.1, schema calculus is another type of shorthand notation used to avoid monolithic descriptions. The schema calculus operators produce expressions which combine existing schemas into new, more complex descriptions (35:204). During semantic analysis, the resultant combinations must be dissected, expanded, and checked for type compatibility.

   - *PreCondition Calculations* - In *Z*, a *precondition* is defined as a predicate that defines the domain of an operation that represents a relation between a start and resulting state space (35:216). Expansion of the schema calculus hiding (\) operator requires precondition calculations to uncover any operation schema defects, i.e., *false* preconditions.

   - *Type Checking* - Type checking ensures that operators used in expressions are not applied to incompatible operands (1). Although existing type-checkers are suitable for stand-alone verification, the incorporation of an internal type-checker would provide a continuous compilation path within the REFINE environment.

5. *Identify Theorem Proving Tasks* - During the refinement of a specification into executable code, it is vital that the ability to reason formally be retained. Mathematical theorems and proofs enable formal reasoning and should be used to demonstrate con-

sistency and completeness (32:145). The availability of automated theorem provers presents an opportunity for performing incremental verifications of the specification throughout the refinement process.

To capitalize on these resources, further research should identify potential theorem proving tasks. Intuitively, these tasks could be correlated to the two major phases of refinement: data and operation. In the data refinement phase, a concrete data type must be constructed to simulate each abstract one. The abstract state must be *adequately represented* by the concrete types and each abstract operation must be correctly recast over the new state (35:243). Potential theorem proving tasks for this phase include:

(a) Verify that each abstract variable can be *retrieved* from the concrete ones.

(b) Verify that the concrete initial state describes an abstract counterpart.

(c) Verify that each concrete operation has a function that at least encompasses its abstract partner.

(d) Verify that each concrete operation produces behavior that its abstract partner would be capable of producing.

In the operation refinement phase, each concrete operation is transformed into an implementation algorithm using rules and checks to guarantee correctness. Possible theorem proving tasks at this stage include:

(a) Verify that the strength of the predicates is not diminished.

(b) Verify that variable bindings are maintained.

6. *Complete the Execution Framework* - To complete the execution framework the following tasks are required:

   - *Additional Mapping Analysis* - Required to complete the translation of axiomatic equations in LARCH and schema predicates in *Z* into REFINE functions. The main emphasis for this task centers around identifying the appropriate preconditions and postconditions to establish the corresponding REFINE transform operations.

   - *Execution Translation* - Complete translation of the function and rule classes is the final task in completing the execution framework. This task produces an executable program of a formally specified problem.

   - *System Testing* - Necessary to validate the entire formal compilation process. The established validation domains, counter and fuel tank, should be used to demonstrate and validate the execution of the specified behavior. Additionally, in order to satisfy the constraints of completeness and consistency, larger and more robust domains should be evaluated.

   - *System Verification* - Required to verify the consistency of the execution translation, i.e., *Z* Behavior => REFINE Behavior. For example, theorem proving tasks need to be identified to verify that set-formers in the *Z* specification are consistent with set-formers in the REFINE program.

For a "big picture" perspective of these recommendations with respect to the formalized transformation process, Figure 7.2 depicts a visualization of the complete target process model.

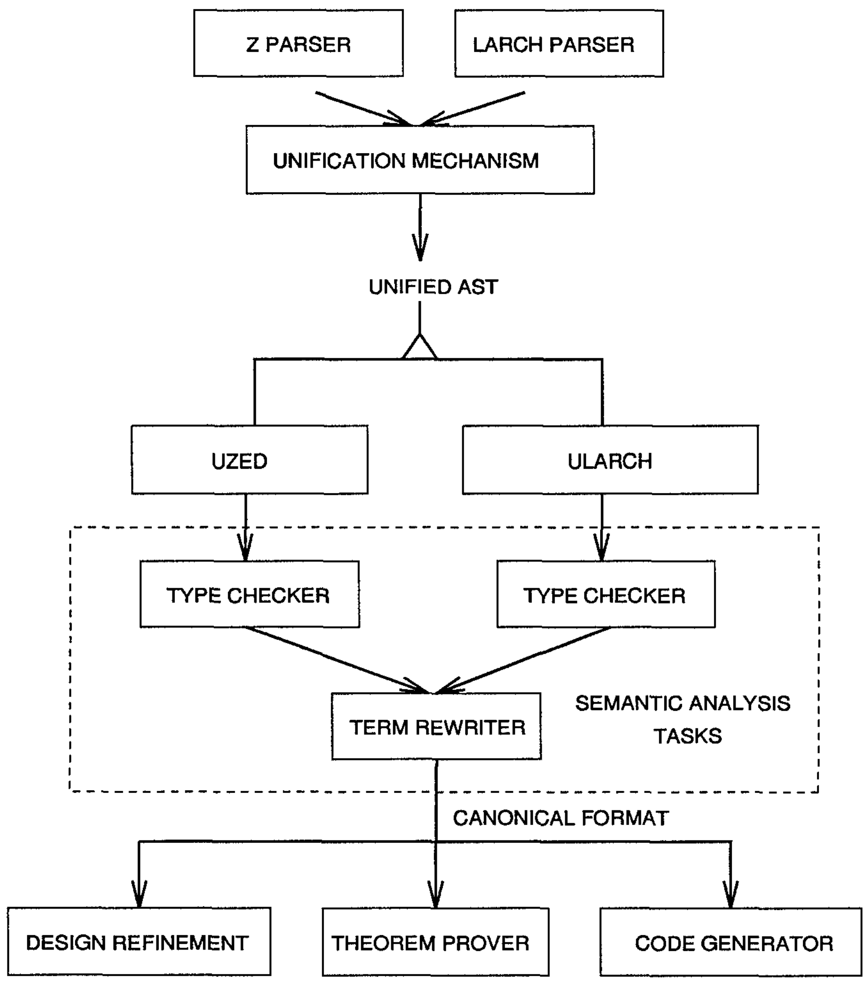

Figure 7.2 Target Transformation Process

7.5 Final Comments

The desirable characteristics of future software development will be an amalgamation of various disciplines and methodologies (24:53). Foremost of these attributes is the combination of formal languages with object-oriented analyses. The formalized object transformation process developed during this thesis effort possesses this attribute and therefore, is a significant milestone toward the establishment of the next generation application composition system. Additionally, this research provides an initial capability for the correct and consistent execution of *Z*, thereby strengthening and extending the software engineer's toolkit.

The Counter Analysis Models were selected as an informal testing domain for the formal object transformation process. This appendix contains the OMT analysis models and the resultant Z specification.

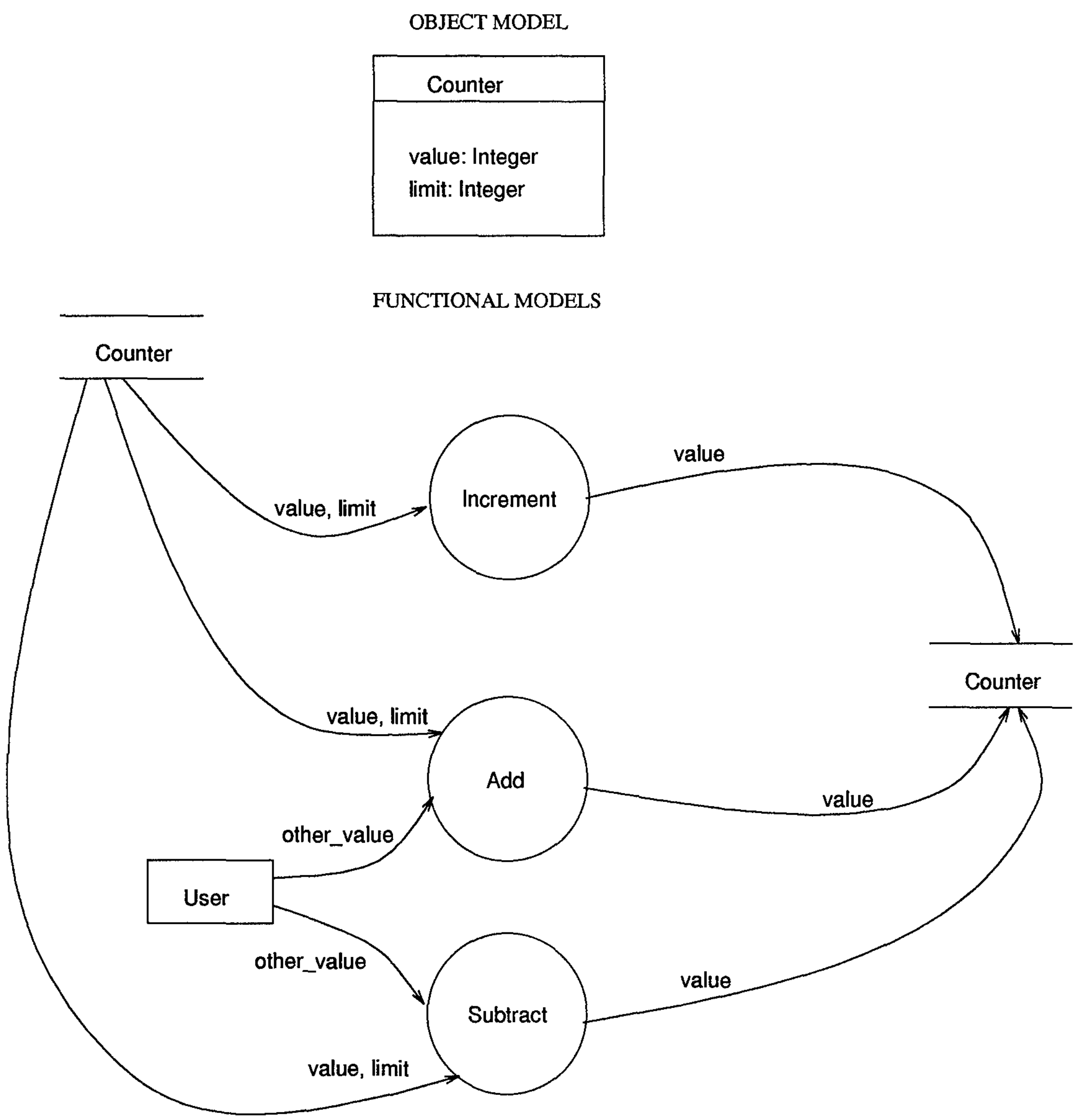

Figure A.1 Counter Object and Functional Models

```
(#>
\documentstyle[fullpage,zed]{article}
  \begin{document}
   \begin{schema}{ Counter }%ObjectTheory
     Value  : \nat\\
     Limit  : \nat
   \where
     Value \leq Limit
   \end{schema}\\

   \begin{schema}{ InitCounter }%StateTheory
     Counter
   \where
     Value = 0\\
     Limit = 100
   \end{schema}\\

   \begin{schema}{ Increment }%FunctionalTheory
     \Delta Counter\\
   \where
     Value < Limit
   \also
     Value' = Value + 1
   \end{schema}\\

   \begin{schema}{ Add }%FunctionalTheory
     \Delta Counter\\
     AddValue?  :  \nat\\
     NewValue!  :  \nat
   \where
     Value + AddValue? \leq Limit
   \also
     Value' = Value + AddValue?\\
     NewValue! = Value'
   \end{schema}\\

   \begin{schema}{ Subtract }%FunctionalTheory
     \Delta Counter\\
     DecValue?  :  \nat\\
     NewValue!  :  \nat
   \where
     Value - DecValue? \geq 0
   \also
     Value' = Value - DecValue?\\
     NewValue! = Value'
   \end{schema}
  \end{document})
```

Appendix B. Validation Specifications

This appendix contains the U*Zed* parser validation specifications. These specifications contain examples of proper grammatical syntax required by the parser. One specific item of importance is the placement of parentheses around the predicates and expressions listed below. These parentheses are necessary to reduce ambiguity during parsing.

1. Predicates - $\wedge$, $\vee$, $\Leftrightarrow$, and $\Rightarrow$.
2. Schema Calculus Expressions - $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, $\backslash$, $\preceq$, and ⨟.
3. In-Generic Expressions - ↔, →, ⇸, ⊂, ⋐, ↠, ⤀, ⤖, ⇻, and ⋐.

B.1 Data Dictionary

```
(#>
\documentstyle[fullpage,zed]{article}
  \begin{document}
   %{\Large\bf Z Specification of a Data Dictionary}

   \begin{zed}
      [ NAME, INFO ]
   \end{zed}\\

   \begin{syntax}
       Response & ::= & Success \\
                & |    & Found \\
                & |    & Overflow \\
                & |    & Redefined \\
                & |    & Undefined \\
                & |    & NotFound
   \end{syntax}\\

   \begin{axdef}
      MaxSize  :  \nat
   \where
      MaxSize \leq 65535
   \end{axdef}\\
```

```
\begin{schema}{ DataDictionary }%ObjectTheory
   Dict: NAME \pfun INFO \\
   Defined: \power NAME
\where
   Defined = \dom Dict \\
   \# Defined \leq MaxSize
\end{schema}\\

DataDictInit \defs [ DataDictionary' | Defined' = \empty ]\\

\begin{schema}{ InsertOK }%FunctionalTheory
    \Delta DataDictionary\\
    Name? : NAME \\
    Info? : INFO \\
    Resp! : Response
\where
    Name? \notin Defined \\
    \# Defined < MaxSize \\
    Dict' = Dict \cup \{ Name? \mapsto Info? \} \\
    Resp! = Success
\end{schema}\\

\begin{schema}{ RemoveOK }%FunctionalTheory
   \Delta DataDictionary \\
   Name? : NAME \\
   Resp! : Response
\where
   Name? \in Defined \\
   Dict' = \{ name? \} \ndres Dict \\
   Resp! = Success
\end{schema}\\

\begin{schema}{ SearchOK }%FunctionalTheory
   \Xi DataDictionary \\
   name? : NAME \\
   info! : INFO \\
   resp! : Response
\where
   name? \in defined \\
   info! = dict~name? \\
   resp! = Found
\end{schema}\\
```

```
\begin{schema}{ InsertOverflow }%FunctionalTheory
   \Xi DataDictionary \\
   Name? : NAME \\
   Info? : INFO \\
   Resp! : Response
\where
  \# Defined = MaxSize \\
  Resp! = Overflow
\end{schema}\\

\begin{schema}{ InsertAlreadyDefined }%FunctionalTheory
   \Delta DataDictionary \\
   Name? : NAME \\
   Info? : INFO \\
   Resp! : Response
\where
   Name? \in Defined \\
   Dict' = Dict \oplus \{ Name? \mapsto Info? \} \\
   Resp! = Redefined
\end{schema}\\

\begin{schema}{ RemoveUndefined }%FunctionalTheory
   \Xi DataDictionary \\
   Name? : NAME \\
   Resp! : Response
\where
   Name? \notin Defined \\
   Resp! = Undefined
\end{schema}\\

\begin{schema}{ SearchUndefined }%FunctionalTheory
   \Xi DataDictionary \\
   Name? : NAME \\
   Info! : INFO \\
   Resp! : Response
\where
   Name? \notin Defined \\
   Resp! = NotFound
\end{schema}\\

  Insert \defs ((InsertOk \lor InsertOverflow) \lor InsertAlreadyDefined)\\

  Remove \defs (RemoveOk \lor RemoveUndefined)\\

  Search \defs (SearchOk \lor SearchUndefined)

\end{document})
```

B.2 Specification of an Aircraft Passenger Listing

The aircraft passenger listing records the passengers aboard an aircraft. There are no seat numbers, passengers are allowed aboard on a first-come-first-served basis, and the aircraft has a fixed capacity. The state of the system is given by the set of people on board the aircraft, and the number of persons on board must never exceed the capacity. There must be operations to allow a person to both board and disembark the aircraft. Additionally, there must be operations to discover the number of persons on board and whether or not a given person is in on board (22).

```
(#>
 \documentstyle[fullpage,zed]{article}
   \begin{document}
    %{\Large\bf Z Specification of an Aircraft Passenger Listing}

    \begin{zed}
       [ PERSON ]
    \end{zed}\\

    \begin{axdef}
       Capacity  : \nat
    \end{axdef}\\

    \begin{schema}{ AirCraft }%ObjectTheory
       OnBoard  :  \power PERSON
    \where
       \# OnBoard \leq Capacity
    \end{schema}\\

    \begin{schema}{ InitAirCraft }%StateTheory
       AirCraft
    \where
       OnBoard = \{\empty\}
    \end{schema}\\

    \begin{schema}{ InitBoard }%FunctionalTheory
       \Delta AirCraft\\
       Person?  :  PERSON
    \where
       Person? \notin OnBoard\\
       \# OnBoard < Capacity\\
       Onboard' =  OnBoard \cup \{ Person? \}
    \end{schema}\\
```

```
\begin{schema}{ InitDisembark }%FunctionalTheory
   \Delta AirCraft\\
   Person?  :  PERSON
\where
   Person? \in OnBoard\\
   Onboard' =  OnBoard \setminus \{ Person? \}
\end{schema}\\

\begin{schema}{ NumberOnBoard }%FunctionalTheory
   \Xi AirCraft\\
   NumOnBoard!  :  \nat
\where
   NumOnBoard! \in \# OnBoard
\end{schema}\\

\begin{syntax}
   ANSWERTYPE ::= Yes | No
\end{syntax}\\

\begin{schema}{ PersonOnBoard }%FunctionalTheory
   \Xi AirCraft\\
   Person? :  PERSON\\
   Reply!  :  ANSWERTYPE
\where
   ((Person? \in OnBoard \land Reply! = Yes) \lor
   (Person? \notin OnBoard \land Reply! = No))
\end{schema}\\

\begin{syntax}
   RESPONSETYPE ::= OK | AlreadyOnBoard | Full | NotOnBoard | TwoErrors
\end{syntax}\\
```

```
\begin{schema}{ BoardingError }%FunctionalTheory
   \Xi AirCraft\\
   Person?    :  PERSON\\
   Response!  :  RESPONSETYPE
\where
   ((((Person? \in OnBoard \land \# OnBoard = Capacity) \land
    Response! = TwoErrors)

    \lor

   ((Person? \in OnBoard \land \# OnBoard < Capacity) \land
    Response! = AlreadyOnBoard))

    \lor

   ((Person? \notin OnBoard \land \# OnBoard = Capacity) \land
    Response! = Full))
\end{schema}\\

Board \defs ((InitBoard \land OKMESSAGE) \lor BoardingError)\\

\begin{schema}{ DisembarkError }%FunctionalTheory
   \Xi AirCraft\\
   Person?    :  PERSON\\
   Response!  :  RESPONSETYPE
\where
   (Person? \notin OnBoard \land Response! = NotOnBoard)
\end{schema}\\

Disembark \defs ((InitDisembark \land OKMESSAGE) \lor DisembarkError)

\end{document})
```

B.3 Car Radio Specification

A digital car radio has three wavebands: medium, long, and UHF. In each band, there is a set of stations (usable wavelengths) and the radio is designed so that it can be tuned only to these. (That is, it is impossible to select an arbitrary wavelength; the operator can select only transmitting stations.) Initially, when the radio is delivered, the lowest station on the medium band is selected. The controls and indicators are described as follows:

1. An MLU button - cycles round the medium, long, and UHF bands.
2. UP/DOWN buttons - change the current station.
3. VOL control - adjusts the volume and acts as an ON/OFF switch.

The MLU button will change to the next band (M-L-U-M) and select the lowest station on that band. The UP/DOWN buttons cause the radio to search for the next station in the current band (U, L or M) either "up" or "down" the band. When either end is reached, the search "wraps" to the other end of the band. The MLU and UP/DOWN buttons have no effect when the radio is turned off (27:146).

```
(#>
 \documentstyle[fullpage,zed]{article}
   \begin{document}
   %{\Large\bf Z Specification of a Car Radio}

    \begin{syntax}
       OnOff ::= ON | OFF
    \end{syntax}\\

    \begin{axdef}
       MaxVol  : \nat_1
    \end{axdef}\\

    \begin{schema}{ Volume }%ObjectTheory
       Volume  :  0 \upto MaxVol\\
       Main    :  OnOff
    \where
       (Main = OFF \iff Volume = 0)\\
       (Main = ON \iff Volume > 0)
    \end{schema}\\

    WaveLength == \nat_1\\
```

```
\begin{axdef}
   Waveband  :  \power (seq Wavelength)
\where
   \forall w : WaveBand \spot (\forall i, j : dom w | i < j \spot wi < wj)
\end{axdef}\\

\begin{axdef}
   Medium, Long, UHF  :  WaveBand
\where
   WaveBand = (Medium, Long, UHF)\\
   Medium \neq Long \neq UHF
\end{axdef}\\

\begin{schema}{ Bands }%ObjectTheory
   Band         :  WaveBand\\
   Station      :  WaveLength\\
   StationNum   :  \nat_1
\where
   Station = Band StationNum\\
   StationNum \leq \#Band
\end{schema}\\

\begin{schema}{ Radio }%ObjectTheory
   RadioVolume   :  Volume\\
   RadioBands    :  Bands
\end{schema}\\

\begin{schema}{ InitRadio }%StateTheory
   Radio
\where
   Main = OFF\\
   StationNum = 1\\
   Station = Medium 1\\
   Band = Medium
\end{schema}\\

NoOp \defs Radio | Main = OFF\\

ChangeVol \defs Radio | (Band' = Band \land Station' = Station)\\

\begin{schema}{ IncreaseVol }%StateTheory
   ChangeVol
\where
   ((Volume < MaxVol \land Volume' = Volume + 1)

    \lor

   (Volume = 0 \land Volume' = Volume))
\end{schema}\\
```

```
\begin{schema}{ DecreaseVol }%StateTheory
   ChangeVol
\where
   ((Volume > 0 \land Volume' = Volume - 1)

    \lor

   (Volume = 0 \land Volume' = Volume))
\end{schema}\\

\begin{schema}{ ChangeStation }%FunctionalTheory
   \Delta Radio
\where
   Main = ON\\
   Volume' = Volume\\
   Main' = Main\\
   Band' = Band
\end{schema}\\

\begin{schema}{ UpStation }%StateTheory
   ChangeStation
\where
   (StationNum < \# Band \implies StationNum' = StationNum + 1)\\
   (StationNum = \# Band \implies StationNum' = 1)
\end{schema}\\

UP \defs (UpStation \lor NoOp)\\

\begin{schema}{ DownStation }%StateTheory
   ChangeStation
\where
   (StationNum > 1 \implies StationNum' = StationNum - 1)\\
   (StationNum = 1 \implies StationNum' = \# Band)
   Volume' = Volume\\
\end{schema}\\

DOWN \defs (DownStation \lor NoOp)\\

\begin{schema}{ Cycle }%FunctionalTheory
   \Delta Radio
\where
   Main = ON\\
   Volume' = Volume\\
   Main' = Main\\
   StationNum' = 1\\
   (Band = Medium \implies Band' = Long)\\
   (Band = Long \implies Band' = UHF)\\
   (Band = UHF \implies Band' = Medium)
\end{schema}\\
```

```
 MLU \defs (Cycle \lor NoOP)

\end{document})
```

Appendix C. Fuel Tank Specification OMT Analysis Models

The Fuel Tank Analysis Models were selected as a validation domain for the formal object transformation process. This appendix contains the OMT analysis models and state transition table developed by Hartrum (17), and the resultant *Z* specification.

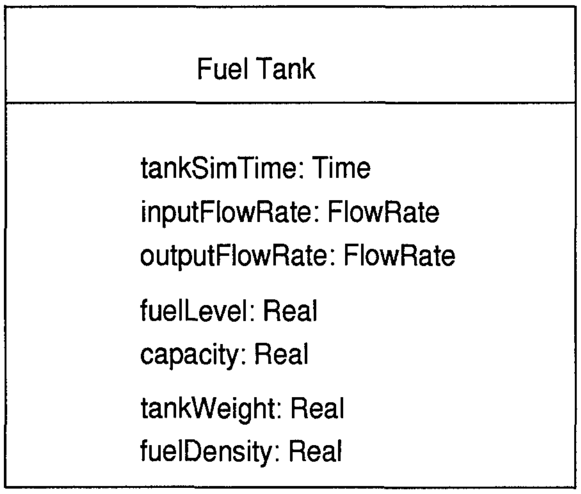

Figure C.1 Fuel Tank Object Model

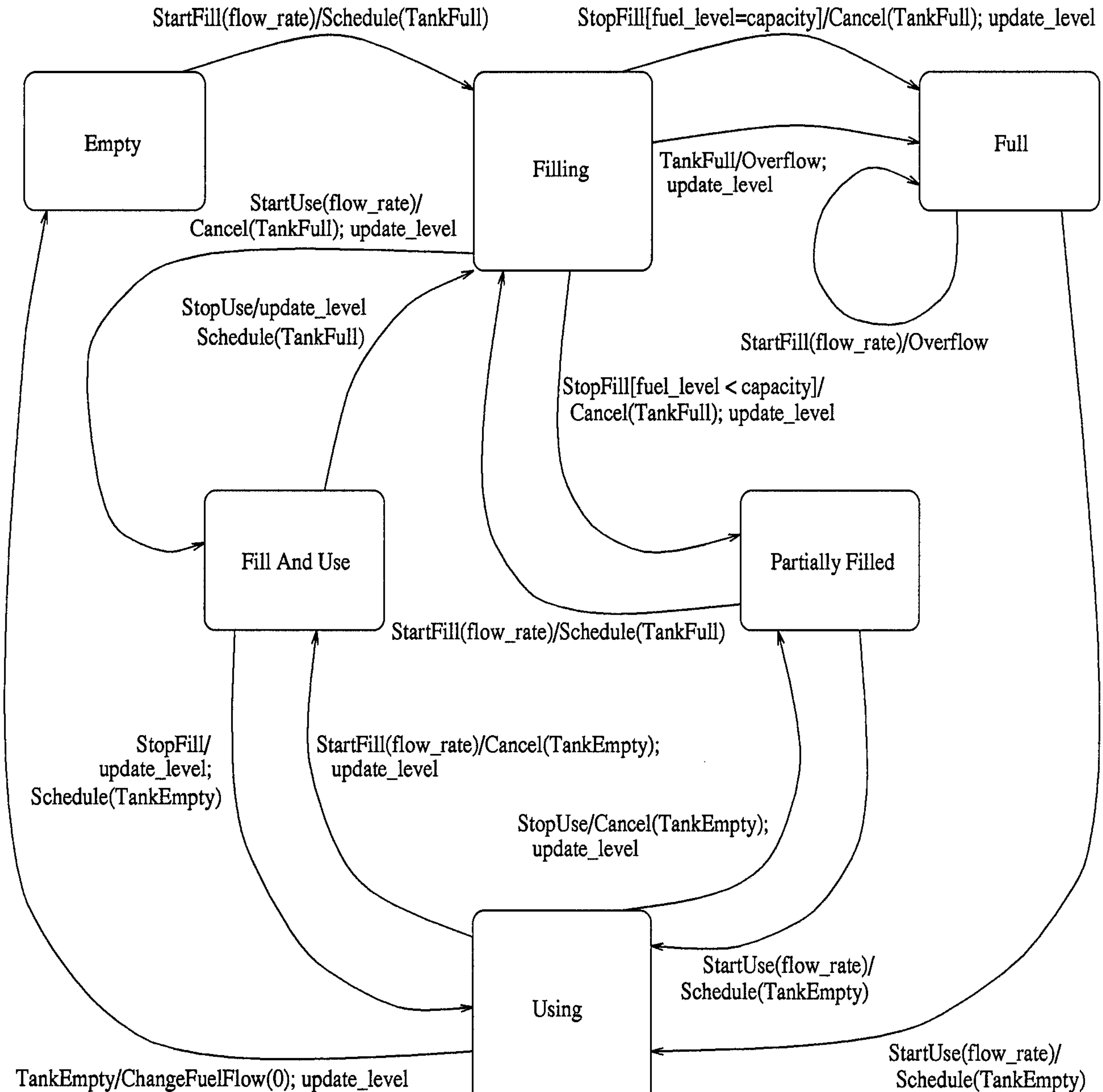

Figure C.2 Fuel Tank Dynamic Model

Table C.1 Fuel Tank State Transition Table.

Current	Event	Guard	Next	Action
Empty	StartFill		Filling	Schedule(TankFull)
Filling	StopFill	$fuel\_level = capacity$	Full	Cancel(TankFull); update_level
Filling	StopFill	$fuel\_level < capacity$	PartiallyFilled	Cancel(TankFull); update_level
Filling	TankFull		Full	Overflow; update_level
Filling	StartUse		FillAndUse	Cancel(TankFull); update_level
Full	StartFill		Full	Overflow
Full	StartUse		Using	Schedule(TankEmpty)
Using	TankEmpty		Empty	ChangeFuelFlow(0); update_level
Using	StopUse		PartiallyFilled	Cancel(TankEmpty); update_level
Using	StartFill		FillAndUse	Cancel(TankEmpty); update_level
PartiallyFilled	StartFill		Filling	Schedule(TankFull)
PartiallyFilled	StartUse		Using	Schedule(TankEmpty)
FillAndUse	StopUse		Filling	Schedule(TankFull); update_level
FillAndUse	StopFill		Using	Schedule(TankEmpty); update_level

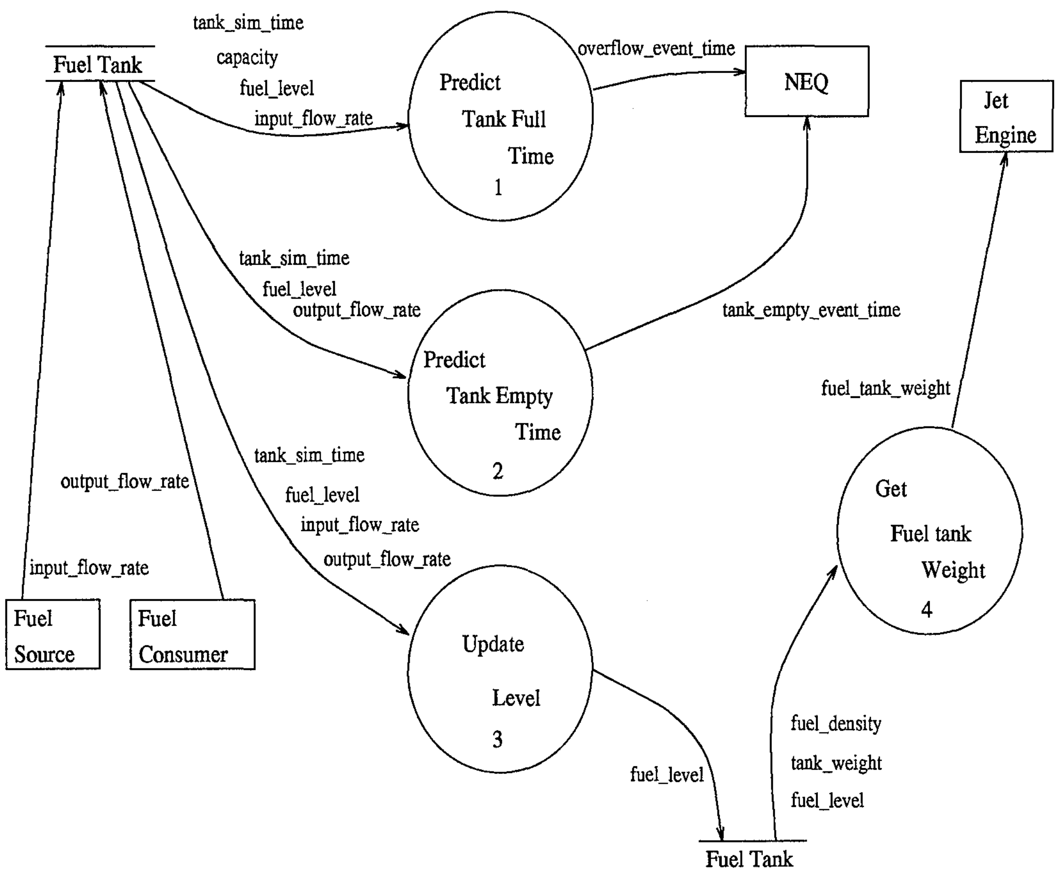

Figure C.3 Fuel Tank Functional Model - 1

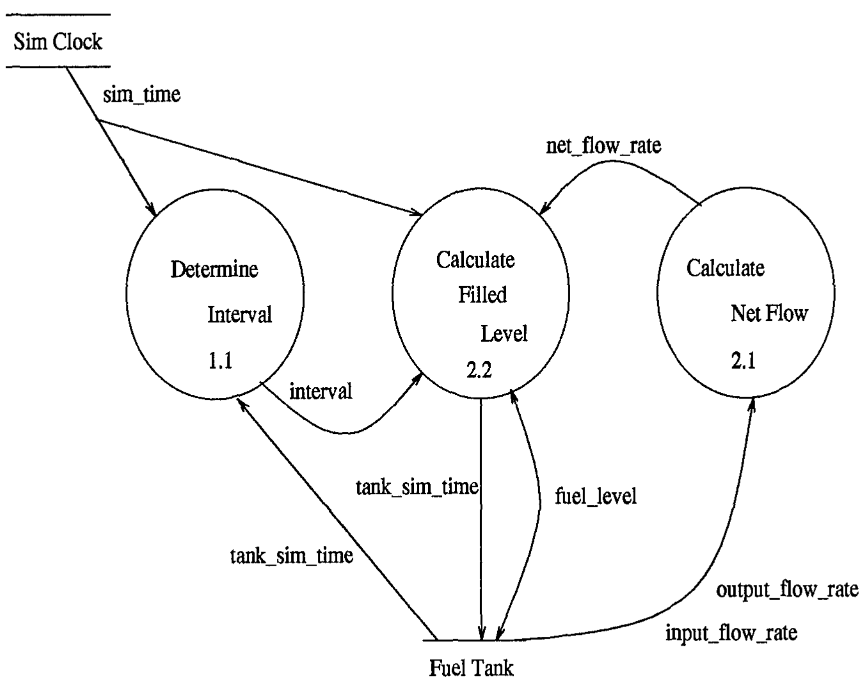

Figure C.4 Fuel Tank Functional Model - 2

```
(#>
\documentstyle[fullpage,zed]{article}
  \begin{document}
   \begin{zed}
      [ SIMTIME ]
   \end{zed}\\

   \begin{schema}{ SimClock }%ObjectTheory
      SimTime   :   SIMTIME
   \end{schema}\\

   \begin{schema}{ FuelTank }%ObjectTheory
      TankSimTime       : SIMTIME\\
      InputFlowRate     : \cal R\\
      OutputFlowRate    : \cal R\\
      FuelLevel         : \cal R\\
      Capacity          : \cal R\\
      TankWeight        : \cal R\\
      FuelDensity       : \cal R\\
      FuelTankWeight    : \cal R
   \where
      FuelLevel \leq Capacity\\
      FuelTankWeight = TankWeight + FuelDensity * FuelLevel
   \end{schema}\\

   \begin{schema}{ Empty }%StateTheory
      FuelTank
   \where
      FuelLevel = 0\\
      InputFlowRate = 0\\
      OutputFlowRate = 0
   \end{schema}\\

   \begin{schema}{ PartiallyFilled }%StateTheory
      FuelTank
   \where
      FuelLevel > 0\\
      FuelLevel < Capacity\\
      InputFlowRate = 0\\
      OutputFlowRate = 0
   \end{schema}\\
```

```
\begin{schema}{ Full }%StateTheory
   FuelTank
\where
   FuelLevel = Capacity\\
   InputFlowRate = 0\\
   OutputFlowRate = 0
\end{schema}\\

\begin{schema}{ Filling }%StateTheory
   FuelTank
\where
   FuelLevel \geq 0\\
   FuelLevel \leq Capacity\\
   InputFlowRate > 0\\
   OutputFlowRate = 0
\end{schema}\\

\begin{schema}{ Using }%StateTheory
   FuelTank
\where
   FuelLevel \geq 0\\
   FuelLevel \leq Capacity\\
   InputFlowRate = 0\\
   OutputFlowRate > 0
\end{schema}\\

\begin{schema}{ FillAndUse }%StateTheory
   FuelTank
\where
   FuelLevel \geq 0\\
   FuelLevel \leq Capacity\\
   InputFlowRate > 0\\
   OutputFlowRate > 0
\end{schema}\\

\begin{schema}{ DetermineInterval }%FunctionalTheory
   \Xi FuelTank\\
   \Xi SimClock\\
   Interval!  : SIMTIME
\where
   Interval! = SimTime - TankSimTime
\end{schema}\\
```

```
\begin{schema}{ CalculateFilledLevel }%FunctionalTheory
   \Delta FuelTank\\
   \Xi SimClock\\
   Interval?  : SIMTIME
\where
   FuelLevel' = FuelLevel + Interval? * InputFlowRate\\
   TankSimTime' = SimTime
\end{schema}\\

\begin{schema}{ PredictTankFullTime }%FunctionalTheory
   \Xi FuelTank\\
   OverflowEventTime!  : SIMTIME
\where
   OverflowEventTime! = TankSimTime + Capacity - FuelLevel div InputFlowRate
\end{schema}\\

\begin{schema}{ CalculateNetFlow }%FunctionalTheory
   \Xi FuelTank\\
   NetFlowRate!  : \cal R
\where
   NetFlowRate! = InputFlowRate - OutputFlowRate
\end{schema}\\

\begin{schema}{ CalculateFillUseLevel }%FunctionalTheory
   \Delta FuelTank\\
   \Xi SimClock\\
   NetFlowRate? : \cal R\\
   Interval?  : SIMTIME
\where
   FuelLevel' = FuelLevel + Interval? * NetFlowRate?\\
   TankSimTime' = SimTime
\end{schema}\\

\begin{schema}{ CalculateUsedLevel }%FunctionalTheory
   \Delta FuelTank\\
   \Xi SimClock\\
   interval?  : SIMTIME
\where
   FuelLevel' = FuelLevel - Interval? * OutputFlowRate\\
   TankSimTime' = SimTime
\end{schema}\\

\begin{schema}{ PredictTankEmptyTime }%FunctionalTheory
   \Xi FuelTank\\
   TankEmptyEventTime!  : SIMTIME
\where
   TankEmptyEventTime! = TankSimTime + FuelLevel div OutputFlowRate
\end{schema}\\
```

```
 \begin{schema}{ DetermineFuelWeight }%FunctionalTheory
    \Xi FuelTank\\
    FuelWeight!  : \cal R
 \where
    FuelWeight! = FuelLevel * FuelDensity
 \end{schema}\\

 \begin{schema}{ CalculateTotalWeight }%FunctionalTheory
    \Xi FuelTank\\
    FuelWeight?  :  \cal R\\
    FuelTankWeight!  :  \cal R
 \where
    FuelTankWeight! = FuelWeight? + TankWeight
 \end{schema}

\end{document})
```

Appendix D. Unified Domain Model

This appendix contains the object-oriented domain models for the unified abstract framework. These models identify the common object classes shared by LARCH and *Z*. The common object classes promote a uniform interface for formalizing object-oriented analysis models and are the foundation for establishing a canonical representation for formal specification languages. Each language's "dialect" (i.e., the variances of the language from the common core) is depicted in the figures as specialized object classes.

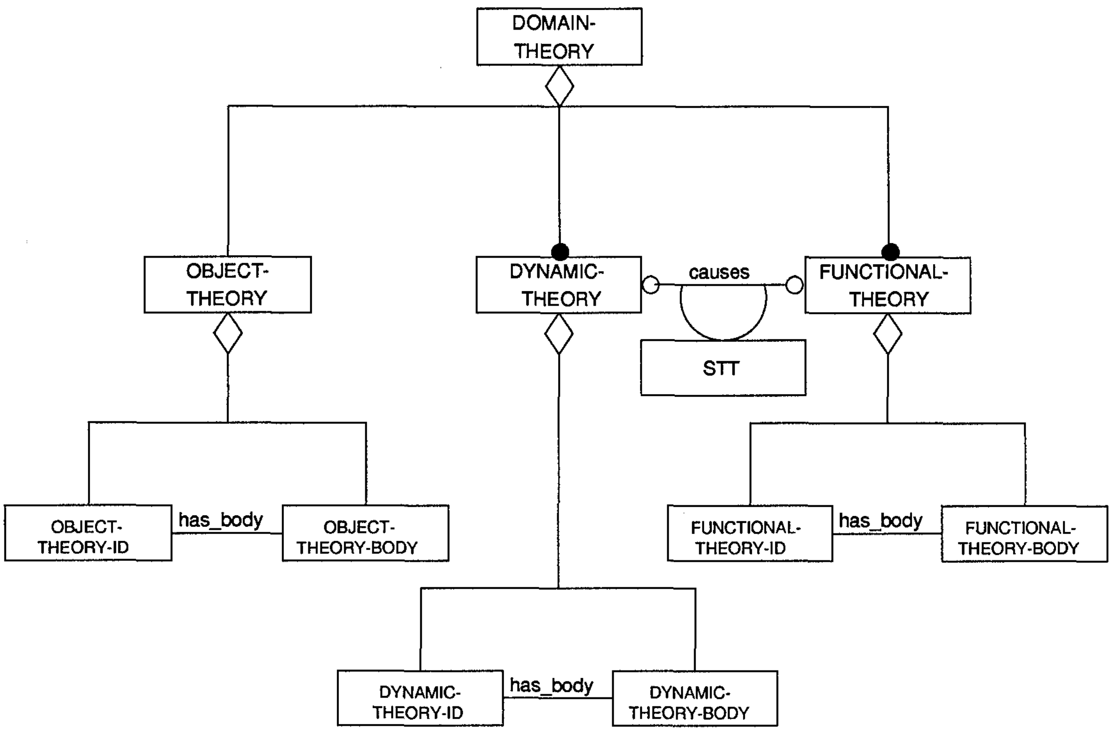

Figure D.1 Unified Domain Theory Model

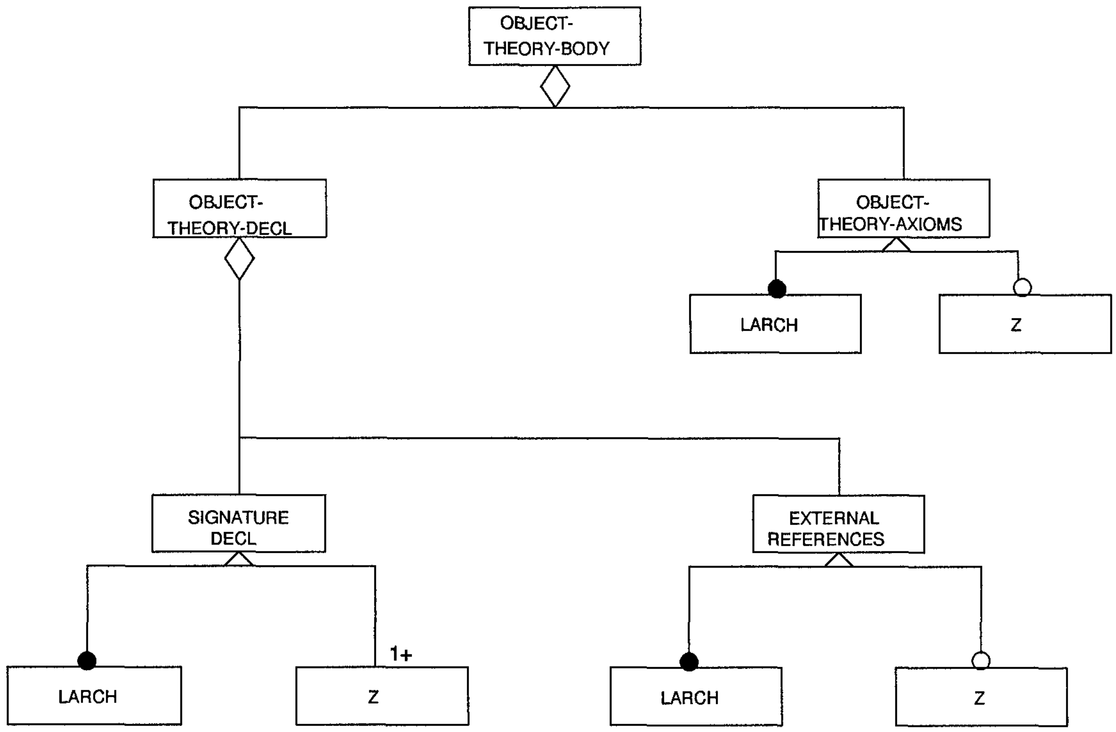

Figure D.2 Unified Object Theory Body

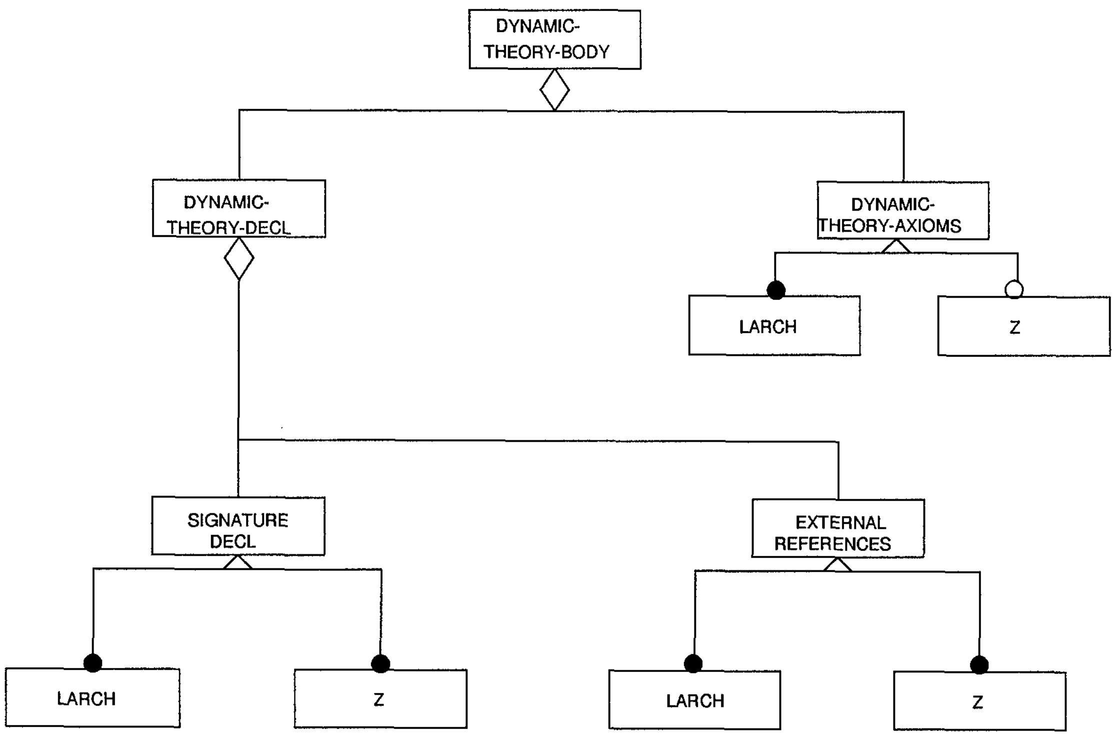

Figure D.3 Unified Dynamic Theory Body

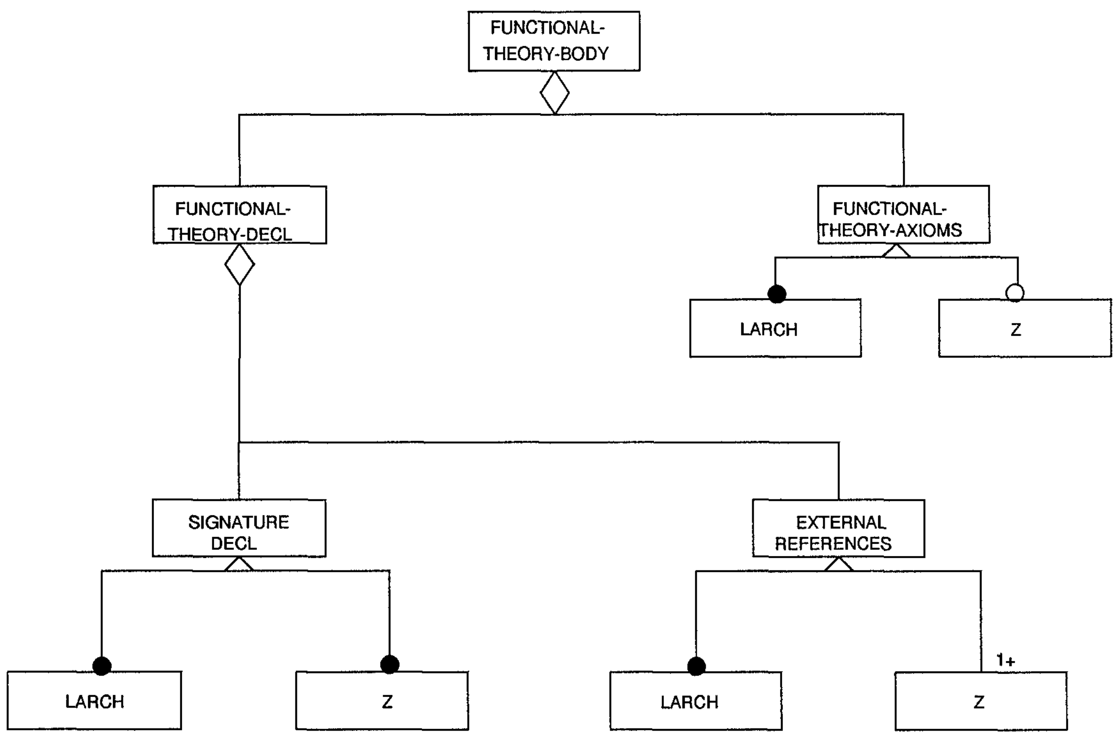

Figure D.4 Unified Functional Theory Body

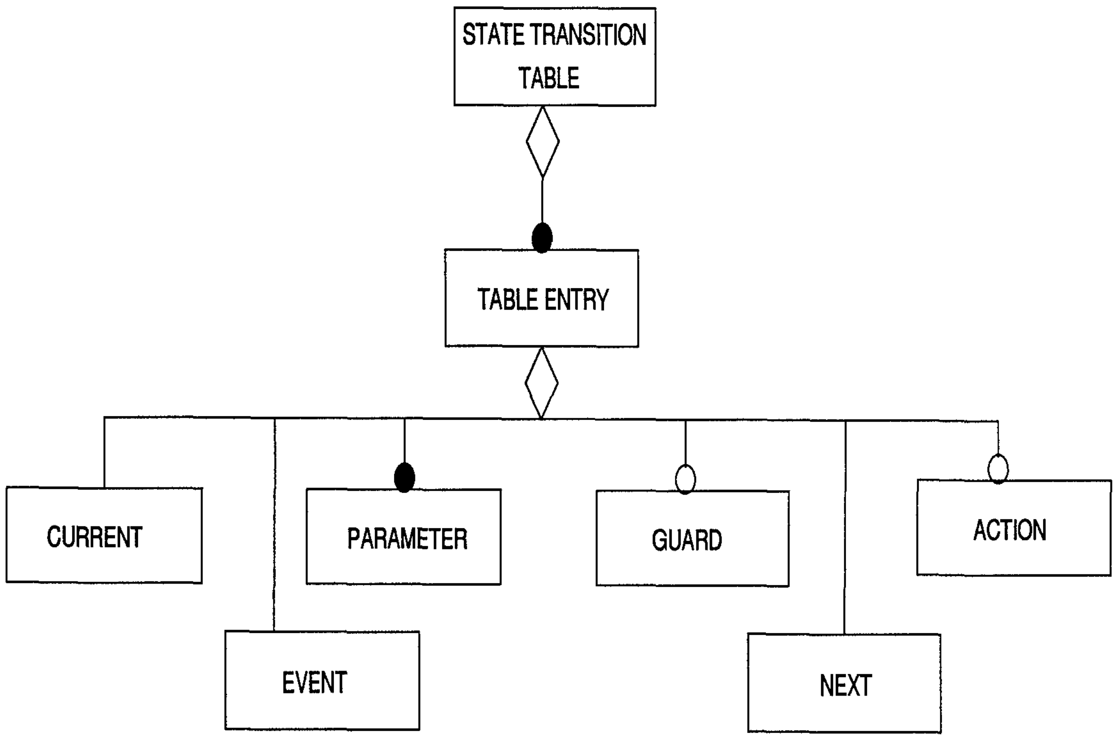

Figure D.5 State Transisition Table

Appendix E. REFINE *Code for the Unified Domain Model*

This appendix contains the REFINE code for the unified abstract framework depicted in Appendix D. This code demonstrates the object-based nature of the REFINE specification language. The objects and attributes specified in REFINE are modular and easily extensible using the notion of subclasses and tree-attributes. The flexibility of this specification language makes future enhancements to the unified abstract model straightforward.

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: unified-dm.re

Description:  The following specification defines the Unified Domain Model's
objects, attributes, and abstract syntax tree.  The unified domain model was
constructed by analyzing the separate abstract syntax trees (AST) of the Larch
and Z languages.
||#

% ----------------------------------------------------------------------------
% Unified Domain Model Object Class Definitions
% ----------------------------------------------------------------------------

var Unified-Object          : object-class subtype-of user-object

var DomainTheory            : object-class subtype-of Unified-Object

var DomainTheoryTypes       : object-class subtype-of Unified-Object
  var ObjectTheory          : object-class subtype-of DomainTheoryTypes
  var DynamicTheory         : object-class subtype-of DomainTheoryTypes
  var FunctionalTheory      : object-class subtype-of DomainTheoryTypes

var TheoryId                : object-class subtype-of Unified-Object
  var ObjectTheoryId        : object-class subtype-of TheoryId
  var DynamicTheoryId       : object-class subtype-of TheoryId
  var FunctionalTheoryId    : object-class subtype-of TheoryId

var TheoryBody              : object-class subtype-of Unified-Object
  var ObjectTheoryBody      : object-class subtype-of TheoryBody
  var DynamicTheoryBody     : object-class subtype-of TheoryBody
  var FunctionalTheoryBody  : object-class subtype-of TheoryBody

var TheoryDecl              : object-class subtype-of Unified-Object
  var ObjectTheoryDecl      : object-class subtype-of TheoryDecl
  var DynamicTheoryDecl     : object-class subtype-of TheoryDecl
  var FunctionalTheoryDecl  : object-class subtype-of TheoryDecl
```

```
var SignatureDecl          : object-class subtype-of Unified-Object
var ContextRef             : object-class subtype-of Unified-Object

var TheoryAxioms           : object-class subtype-of Unified-Object
  var ObjectTheoryAxioms    : object-class subtype-of TheoryAxioms
  var DynamicTheoryAxioms   : object-class subtype-of TheoryAxioms
  var FunctionalTheoryAxioms: object-class subtype-of TheoryAxioms

%-----------------------------------------------------------------------
% Unified Model Attribute Declarations for Branches in Tree Structure
%-----------------------------------------------------------------------
var theory-types           : map(DomainTheory, set(DomainTheoryTypes))
                             computed-using theory-types(x) = {}

var theory-id              : map(DomainTheoryTypes, IdName) = {||}
var ot-id                  : map(DomainTheoryTypes, IdName) = {||}
var dt-id                  : map(DomainTheoryTypes, IdName) = {||}
var ft-id                  : map(DomainTheoryTypes, IdName) = {||}

var theory-body            : map(DomainTheoryTypes, TheoryBody) = {||}
var ot-body                : map(DomainTheoryTypes, TheoryBody) = {||}
var dt-body                : map(DomainTheoryTypes, TheoryBody) = {||}
var ft-body                : map(DomainTheoryTypes, TheoryBody) = {||}

var theory-decl            : map(TheoryBody, TheoryDecl) = {||}
var context-refs            : map(TheoryDecl, set(ContextRef))
                             computed-using context-refs(x) = {}

var signature-decl         : map(TheoryDecl, set(SignatureDecl))
                             computed-using signature-decl(x) = {}

var theory-axioms          : map(TheoryBody, set(TheoryAxioms))
                             computed-using theory-axioms(x) = {}

%-----------------------------------------------------------------
% Structure for Abstract Syntax Tree
%-----------------------------------------------------------------
form Define-Tree-Attributes-of-Unified-Specification
  Define-Tree-Attributes('DomainTheory, {'theory-types})&
  Define-Tree-Attributes('ObjectTheory, {'ot-id, 'ot-body})&
  Define-Tree-Attributes('DynamicTheory, {'dt-id, 'dt-body})&
  Define-Tree-Attributes('FunctionalTheory, {'ft-id, 'ft-body})&
  Define-Tree-Attributes('TheoryBody, {'theory-decl, 'theory-axioms})&
  Define-Tree-Attributes('TheoryDecl, {'signature-decl, 'context-refs})
%%  Define-Tree-Attributes('TheoryAxioms, {%% fill in Larch/Z specific})&
%%  Define-Tree-Attributes('ContextRef, {%% fill in Larch/Z specific})&
%%  Define-Tree-Attributes('SignatureDecl, {%% fill in Larch/Z specific})&
```

Appendix F. REFINE *Code for the State Transition Table*

This appendix contains the REFINE code for the State Transition Table (STT) illustrated in Appendix D. A formal domain model and grammar do not exist for the STT because of the limitations in both Z and LARCH to formally specify the entries in the table. Thus, to complete a formal implementation of the transformation process, a grammar and domain model were developed to parse the STT, thereby generating initial objects to be used in the execution framework.

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: stt.re

Description: The following program defines the State Transition Table (STT)
Domain Model and the Grammar.  The STT is part of the OMT analysis models
and is required to create a complete and accurate execution model for the
formal transformation process.  Compilation of the domain model and grammar
generates an STT parser for state transition tables created using the Rumbaugh
method.  Latex specific notations are used.
||#

% -------------------------------------------------------------------
%  STT Class Definitions
% -------------------------------------------------------------------
var Table               : object-class subtype-of user-object
var StateTable          : object-class subtype-of Table
var StateEntry          : object-class subtype-of Table
var Identifier          : object-class subtype-of Table

%--------------------------------------------------------------------
% STT Attribute Declarations
%--------------------------------------------------------------------
var table-name          : map(Table, Identifier) = {||}
var table-label         : map(Table, Identifier) = {||}
var state-entries       : map(Table, seq(StateEntry)) = {||}
var current-state       : map(Table, Identifier) = {||}
var state-event         : map(Table, Identifier) = {||}
var state-parameter     : map(Table, seq(Identifier)) = {||}
var state-guard         : map(Table, Identifier) = {||}
var next-state          : map(Table, Identifier) = {||}
var state-action        : map(Table, Identifier) = {||}
var num-real            : map(Table, real) = {||}
var num-int             : map(Table, integer) = {||}
```

```
%------------------------------------------------------------------
% Structure for Abstract Syntax Tree
%------------------------------------------------------------------
form Define-Tree-Attributes-of-State-Table
  Define-Tree-Attributes('StateTable, {'table-name, 'table-label,
                                       'state-entries}) &
  Define-Tree-Attributes('StateEntry, {'current-state, 'state-event,
                                       'state-parameter,'state-guard,
                                       'next-state, 'state-action})

%------------------------------------------------------------------
% STT Grammar Definition
%------------------------------------------------------------------
!!in-grammar('syntax)

grammar STT
   start-classes StateTable
   file-classes StateTable
   no-patterns

   case-sensitive
   symbol-continue-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789._-=\\$()<>"
   symbol-start-chars
   "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_$("

  productions

   StateTable   ::= ["\\begin{table}[htb]"
     "\\caption{" table-name "}"
     "\\vspace*{.2in}"
     "\\label{" table-label "}"
     "\\centering"
     "\\begin{tabular}{|l|l|l|l||l|l|}"
     "\\hline"
     "Current" "&" "Event" "&" "Parameters" "&" "Guard" "&"
                    "Next" "&" "Action\\\\"
     "\\hline\\hline"
     state-entries + ""
     "\\hline\\hline"
     "\\end{tabular}"
     "\\end{table}"] builds StateTable,

   StateEntry   ::= [current-state "&" state-event "&" state-parameter * ""
                    "&" {state-guard} "&" next-state "&" {state-action}
     "\\\\" {"\\hline"}] builds StateEntry,

   Identifier   ::= [(name | num-int | num-real)] builds identifier
end
```

Appendix G. UZed Domain Model

This appendix contains the graphical models of the U*Z*ed extensions (including the Mathematical Toolkit) to the unified domain model. These models form the basis for future semantic analysis tasks to reduce the language specific notations into a canonical format. This task can be accomplished by extending the unified abstract framework using pattern matching and term rewriting on the LARCH and *Z* specific object classes.

G.1 UZed Extensions

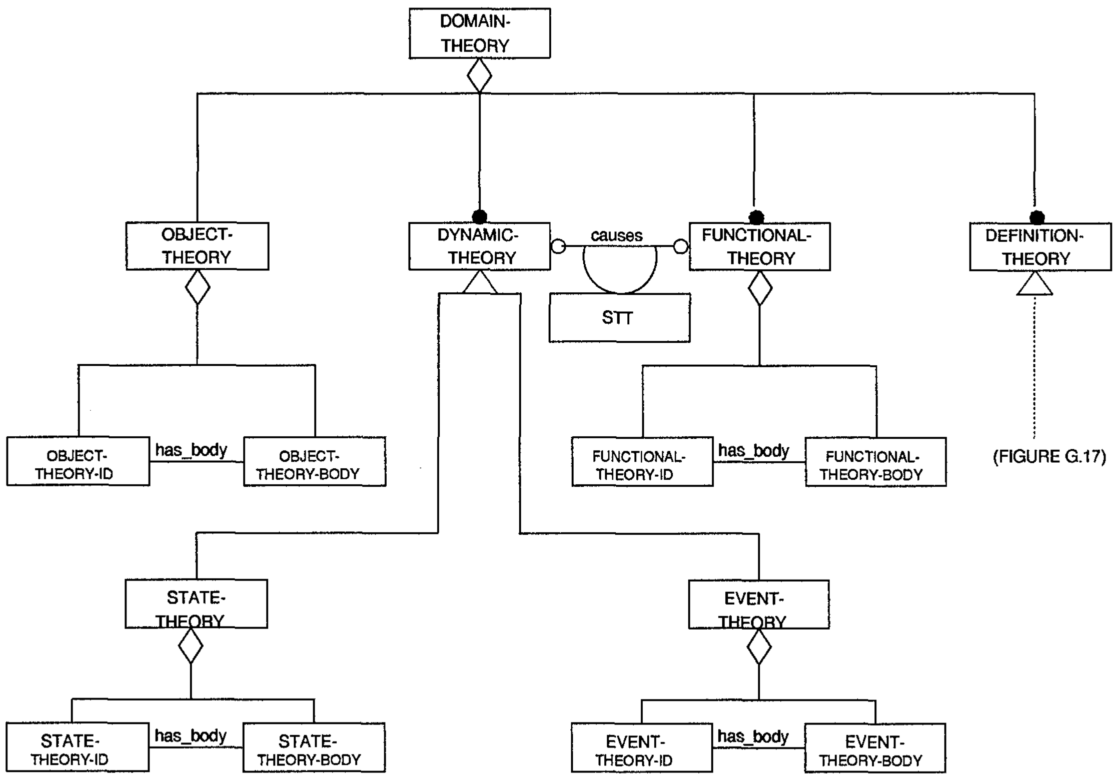

Figure G.1 UZed Domain Theory Model

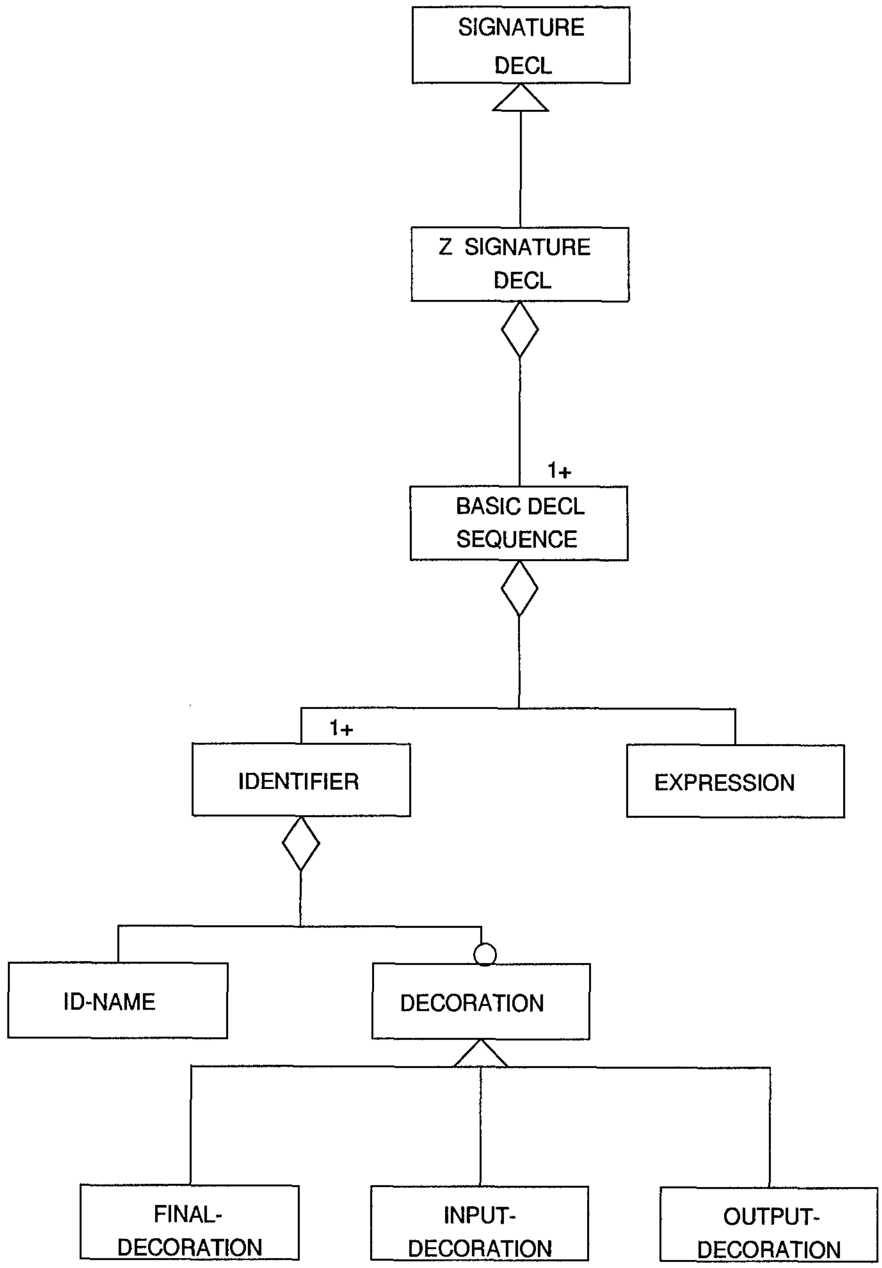

Figure G.2 *UZed* Signature Declaration

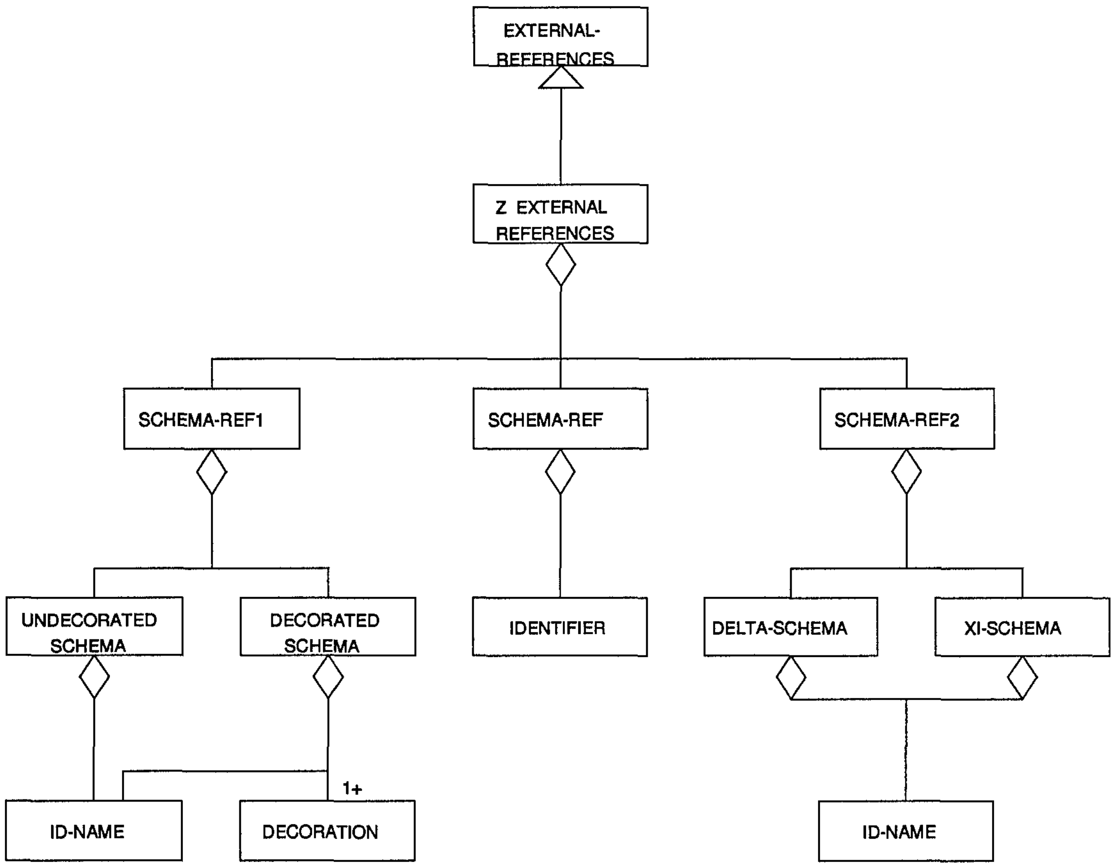

Figure G.3 UZed External References

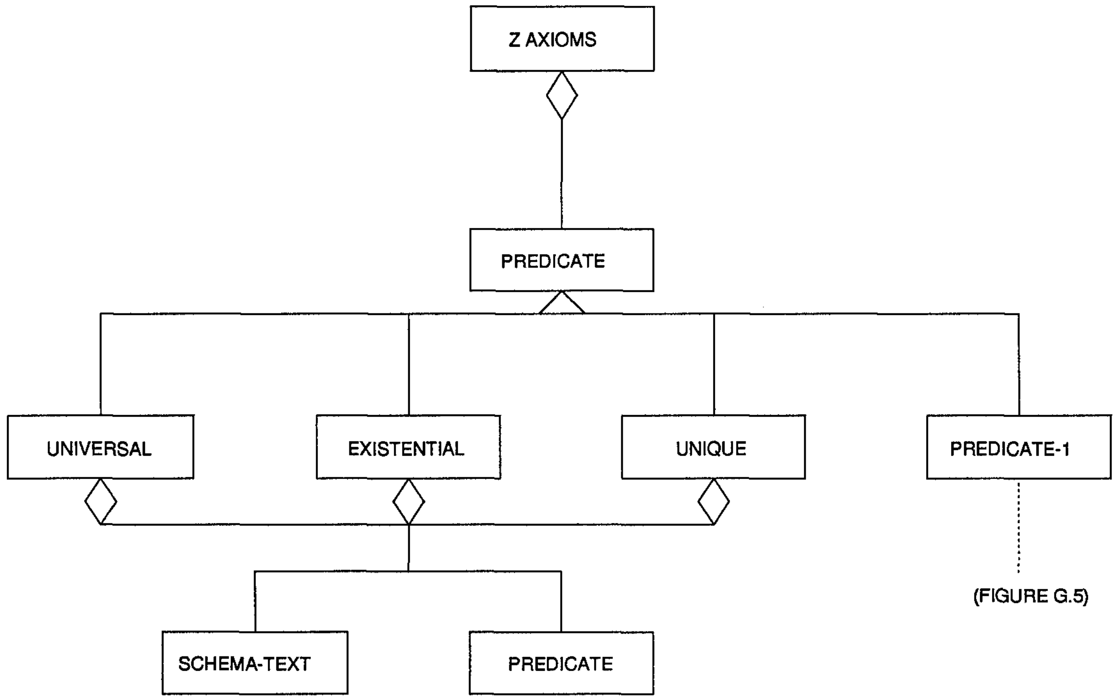

Figure G.4 *Z* Axioms (1 of 5)

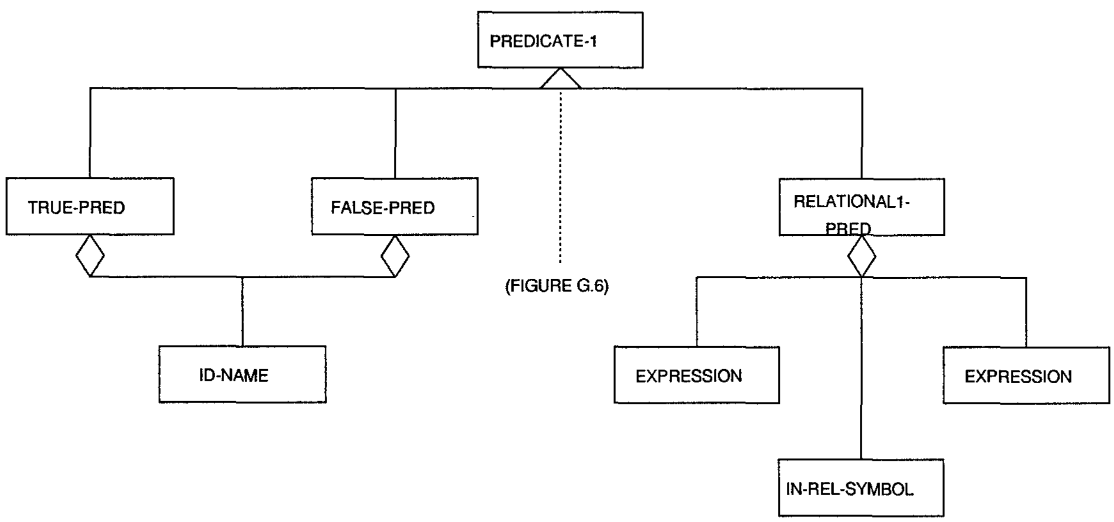

Figure G.5 *Z* Axioms (2 of 5)

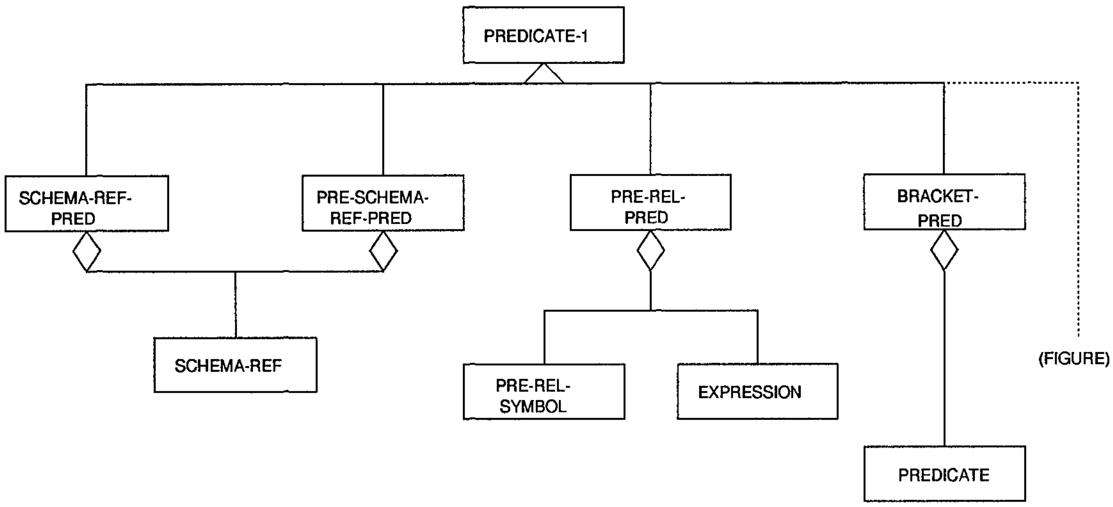

Figure G.6 Z Axioms (3 of 5)

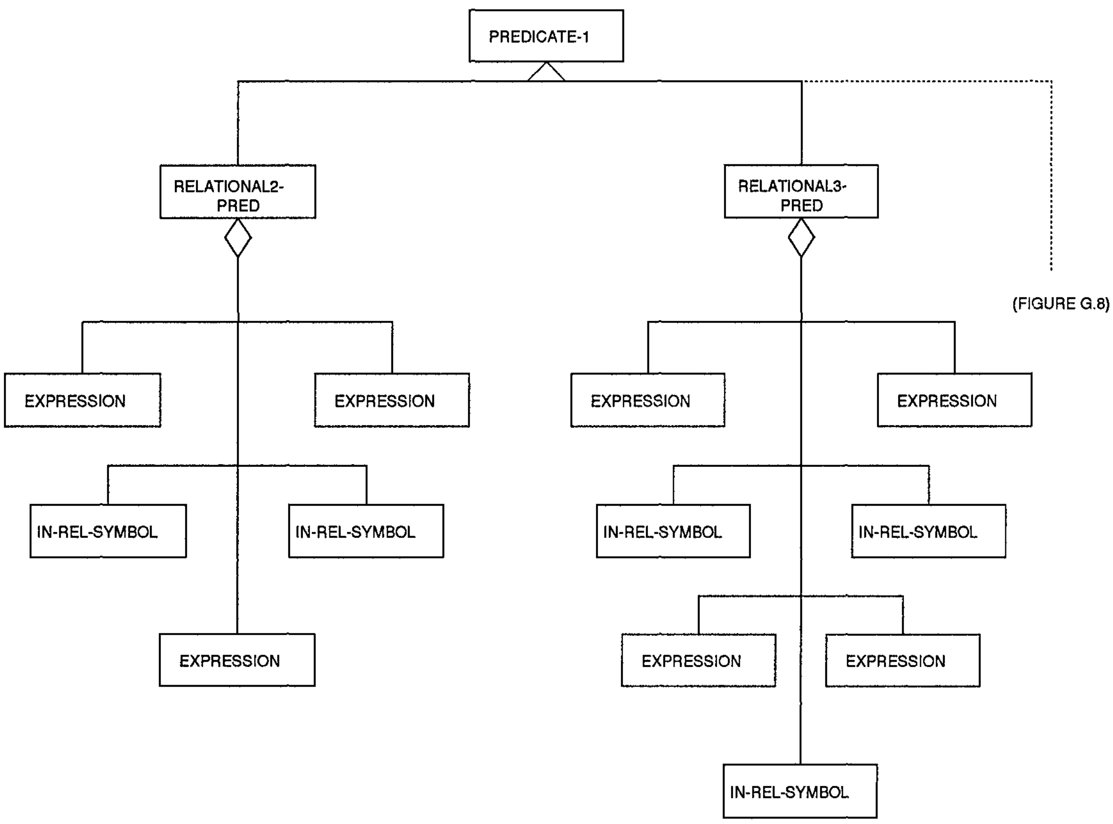

Figure G.7 Z Axioms (4 of 5)

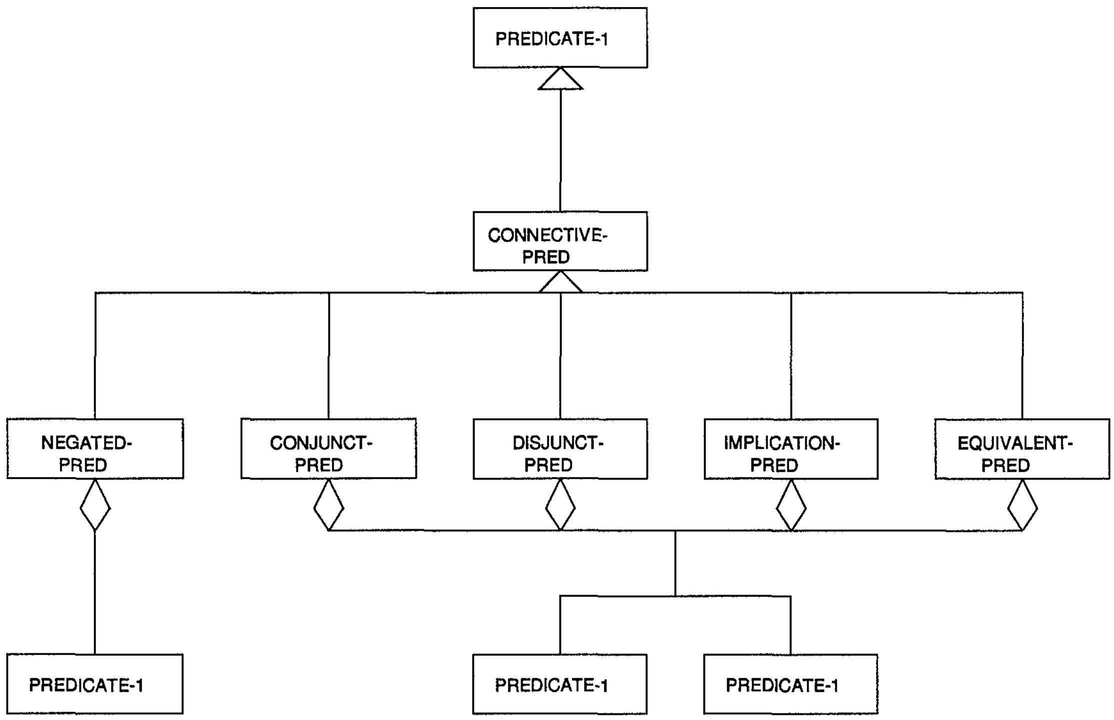

Figure G.8 *Z* Axioms (5 of 5)

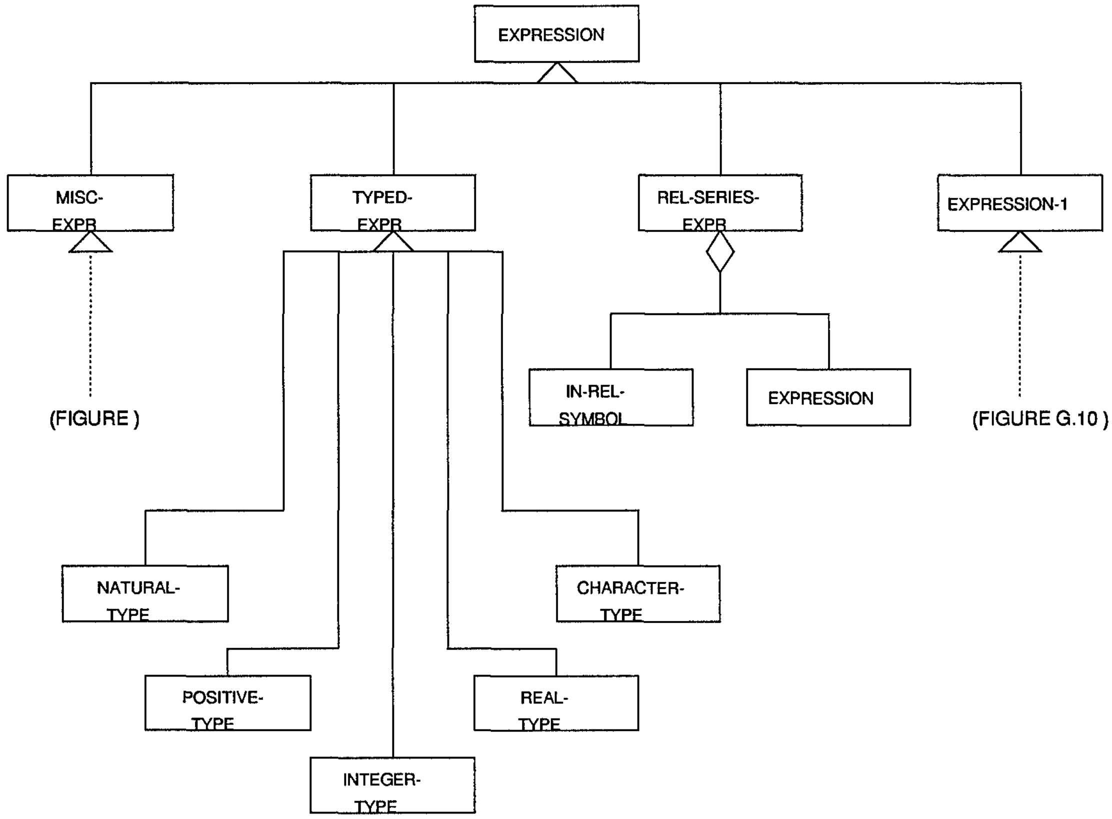

Figure G.9 *Z* Expressions (1 of 7)

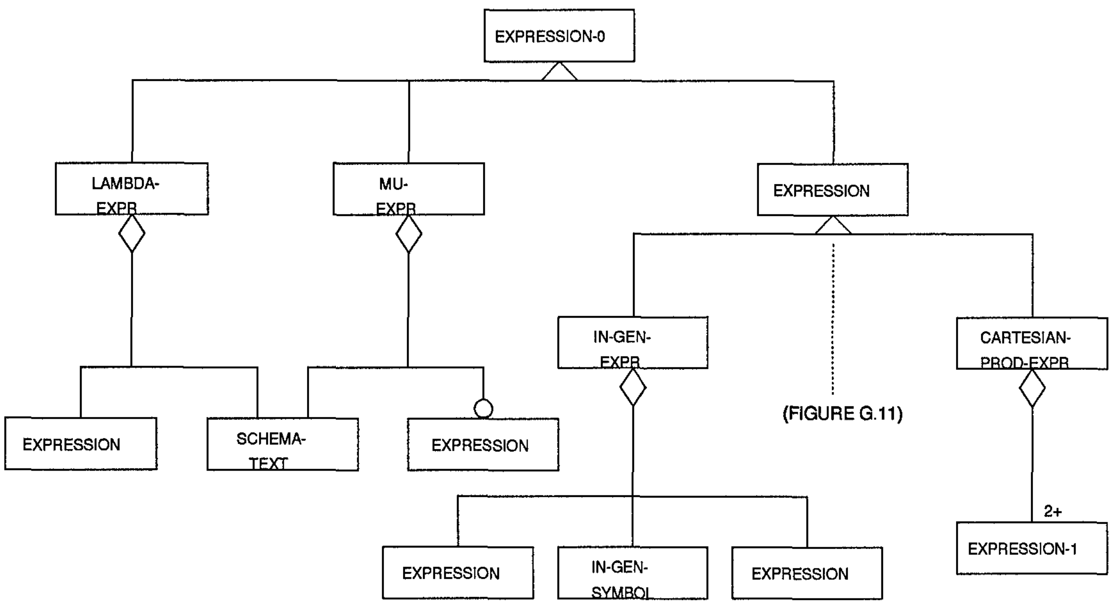

Figure G.10 *Z* Expressions (2 of 7)

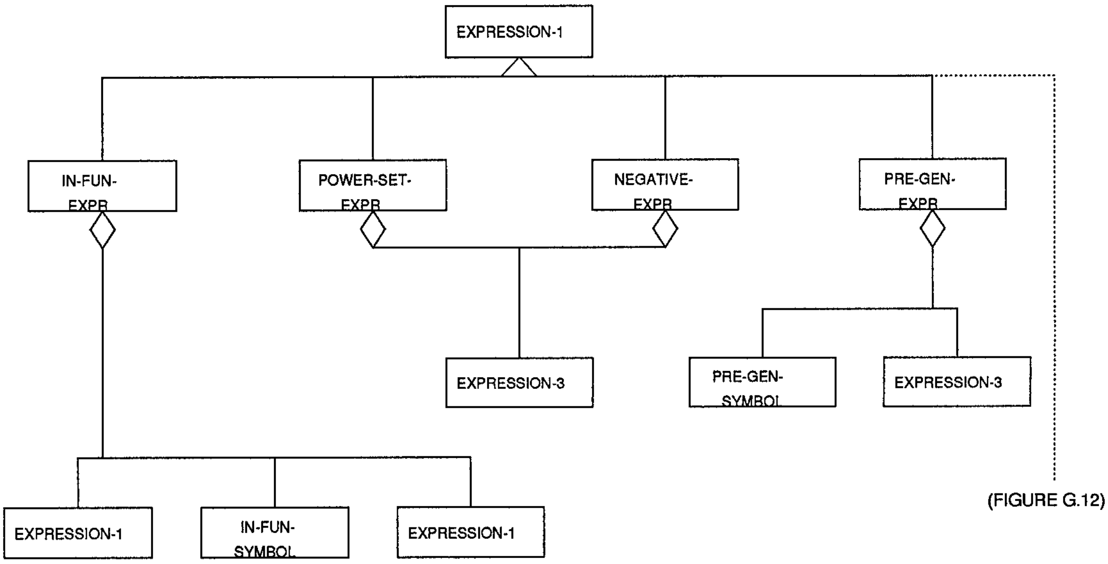

Figure G.11 Z Expressions (3 of 7)

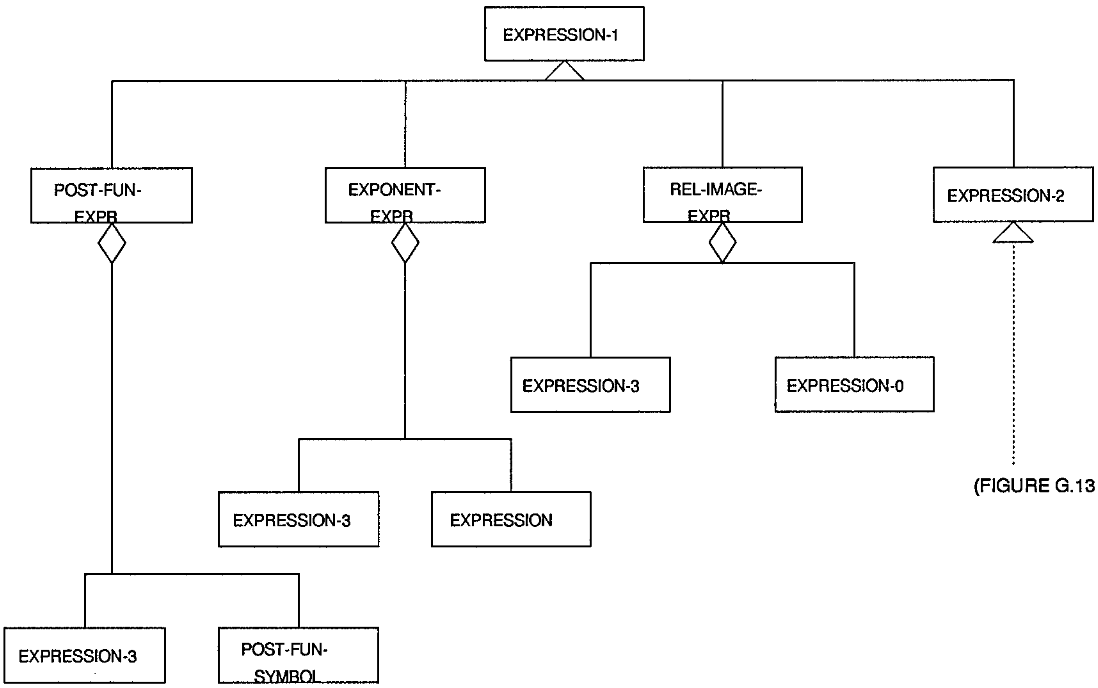

Figure G.12 Z Expressions (4 of 7)

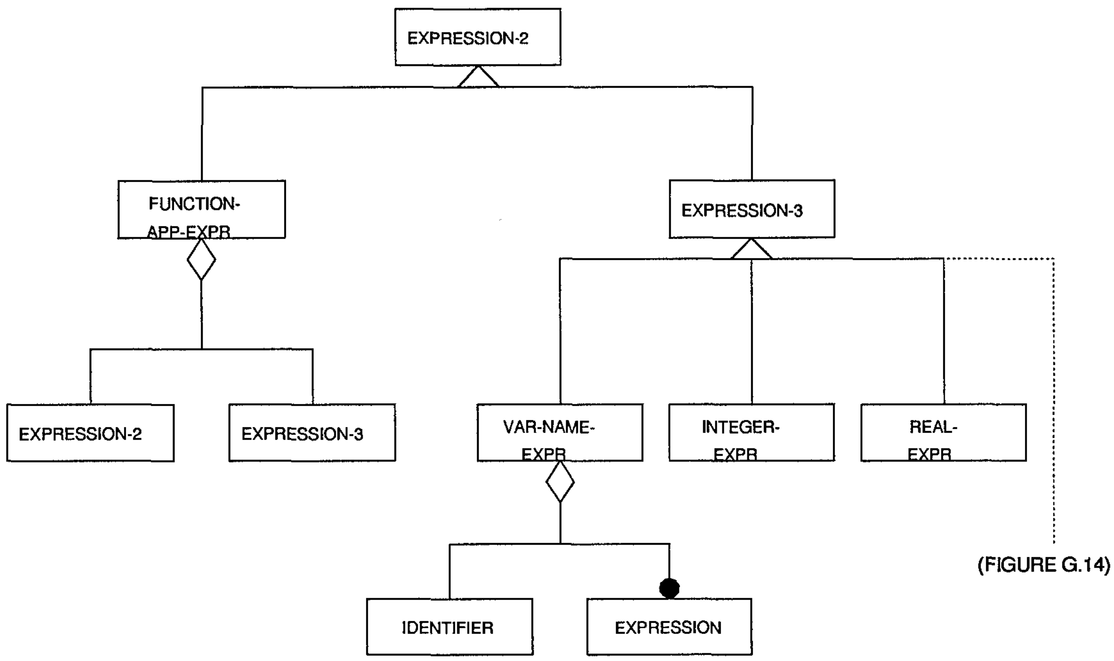

Figure G.13 *Z* Expressions (5 of 7)

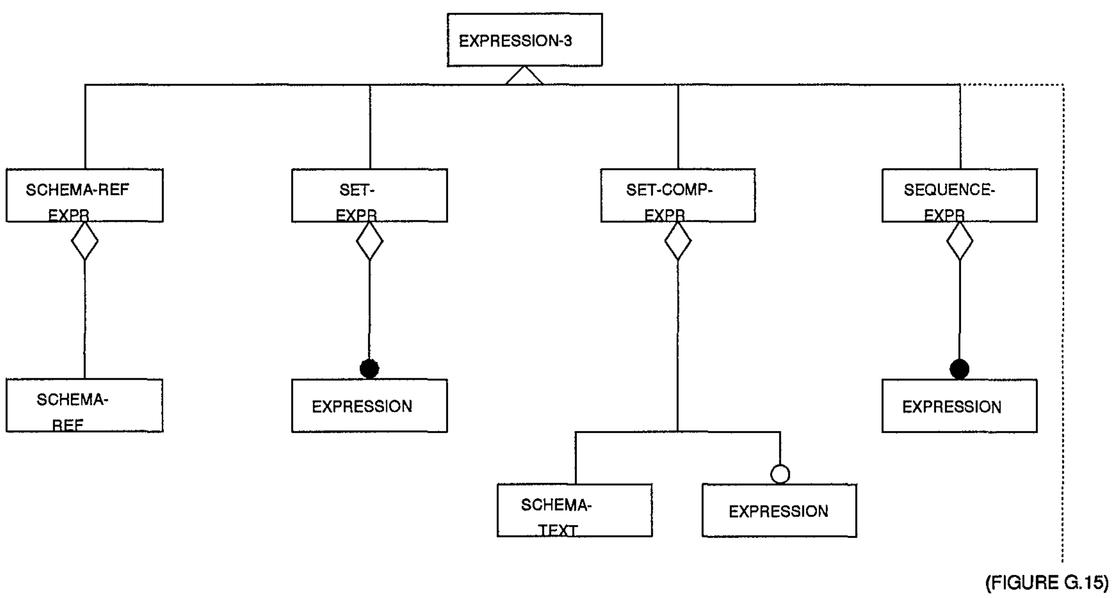

Figure G.14 *Z* Expressions (6 of 7)

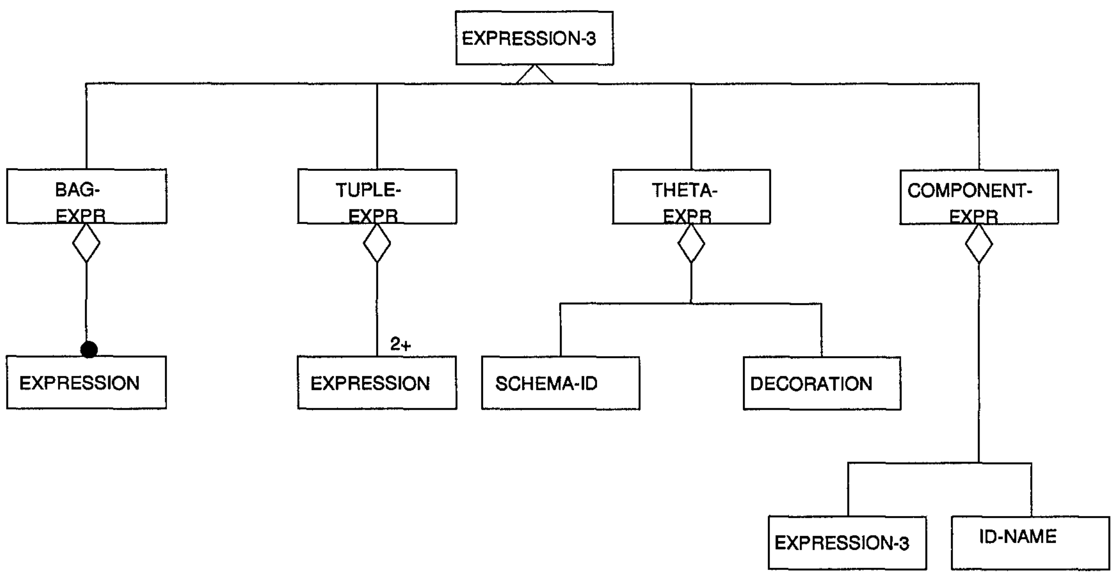

Figure G.15 Z Expressions (7 of 7)

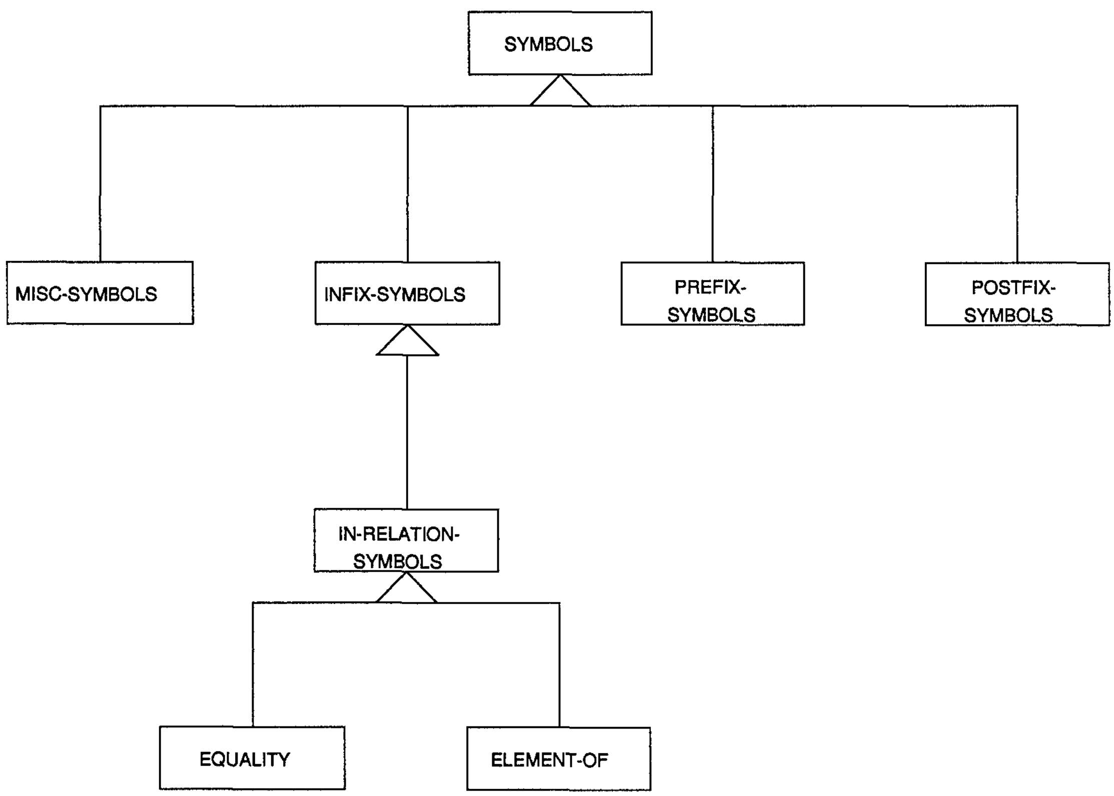

Figure G.16 Core *Z* Symbols

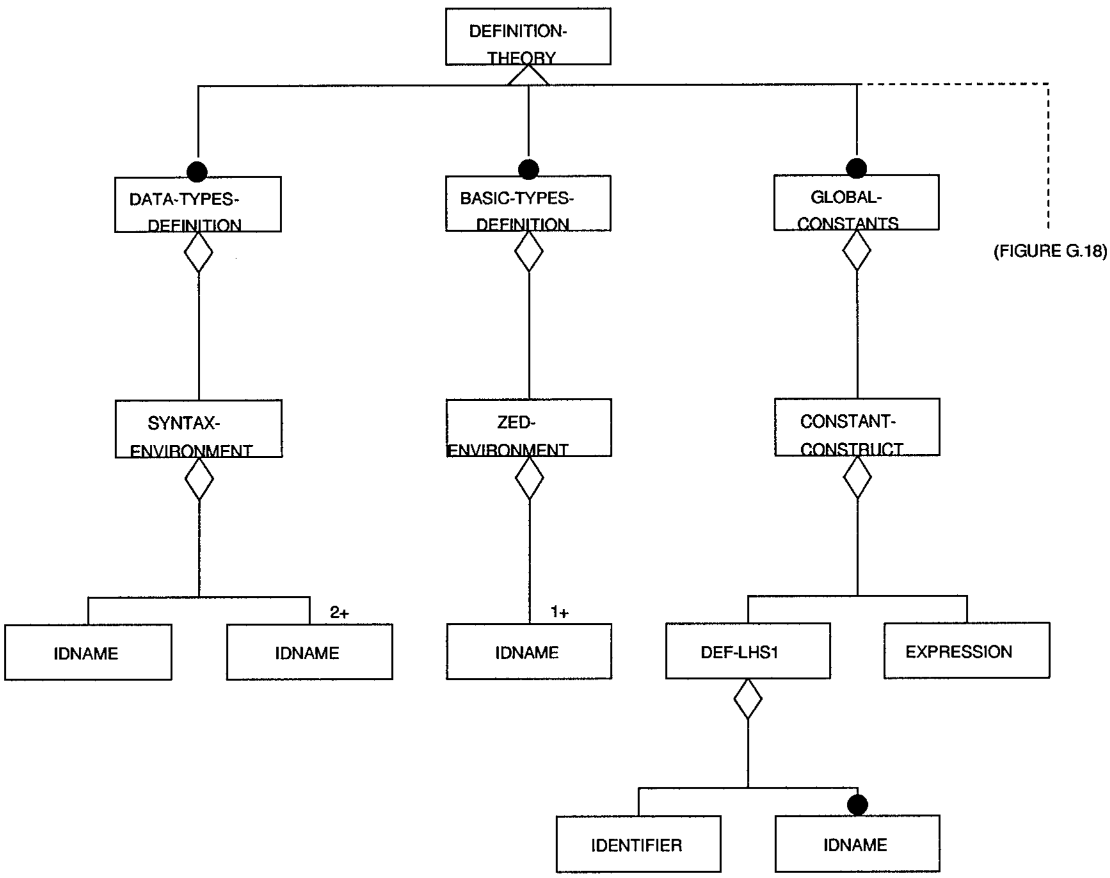

Figure G.17 Definition Theory Model (1 of 3)

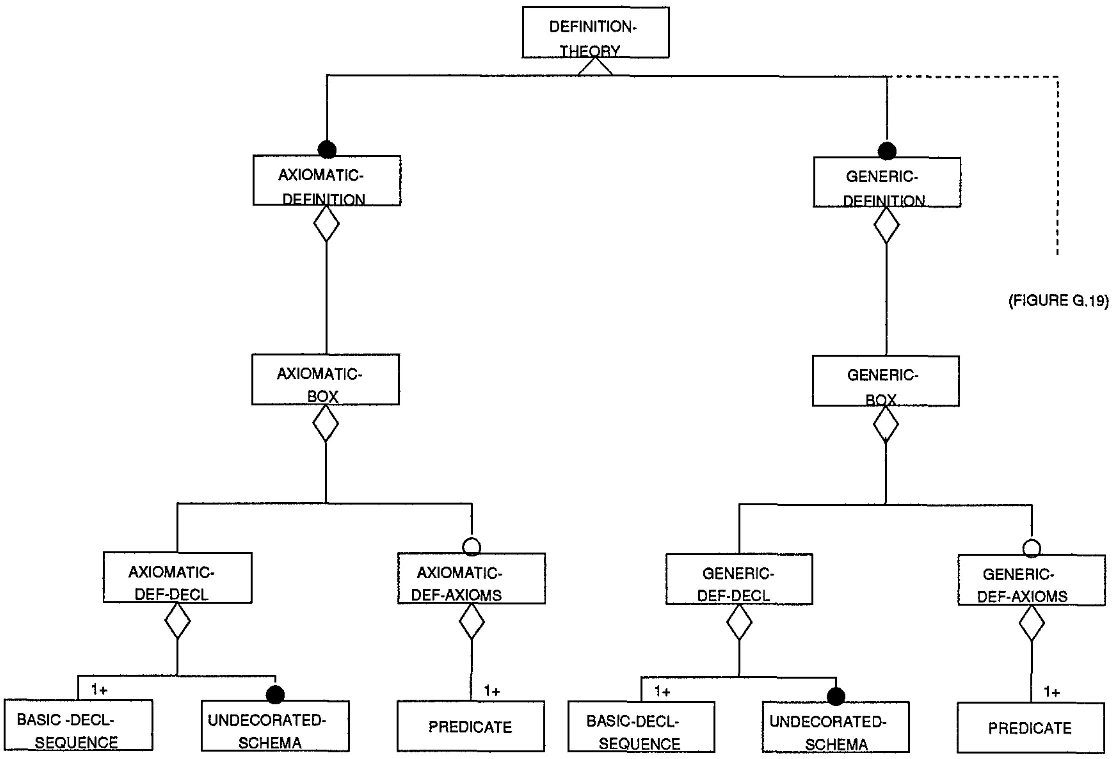

Figure G.18 Definition Theory Model (2 of 3)

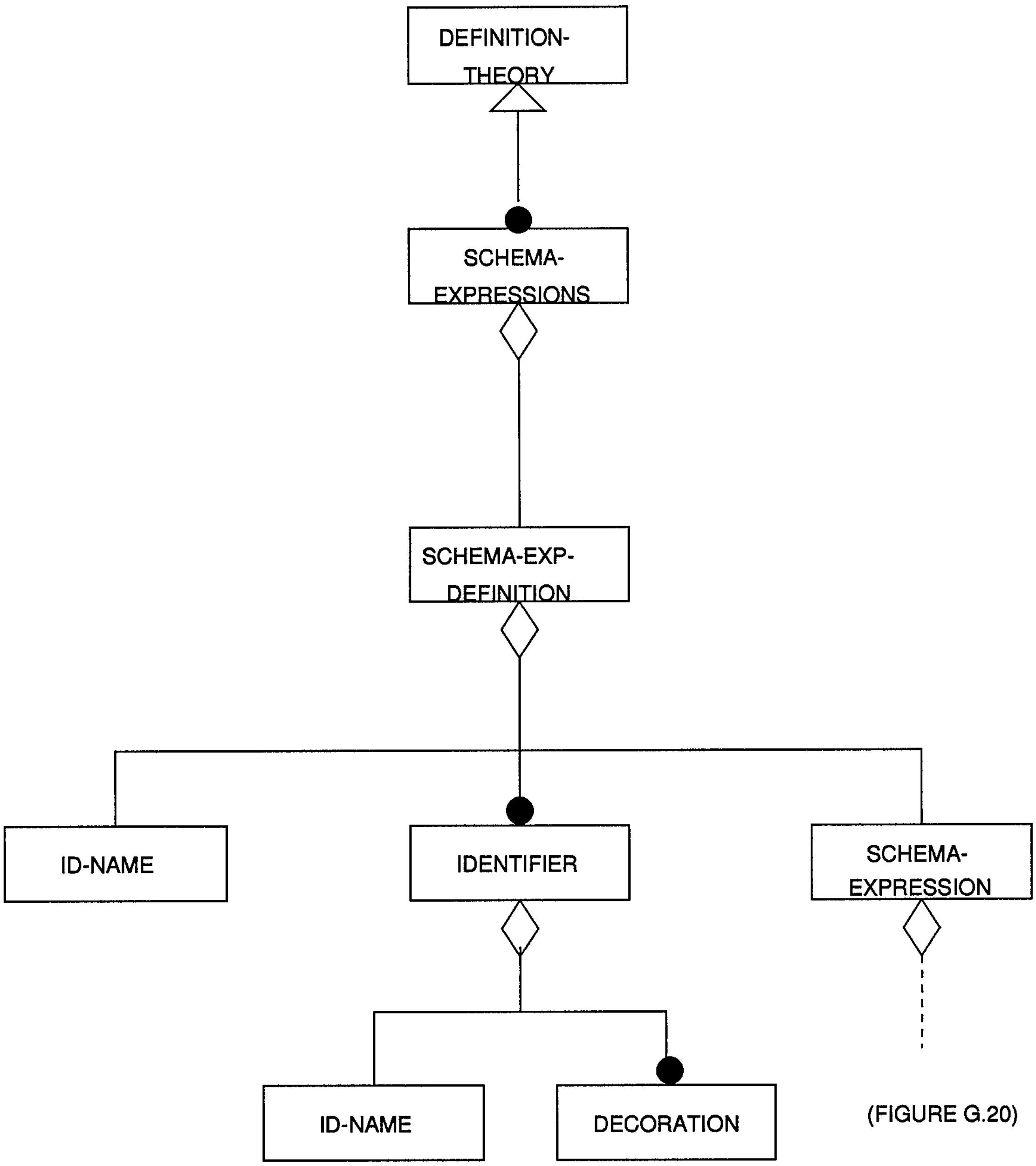

Figure G.19 Definition Theory Model (3 of 3)

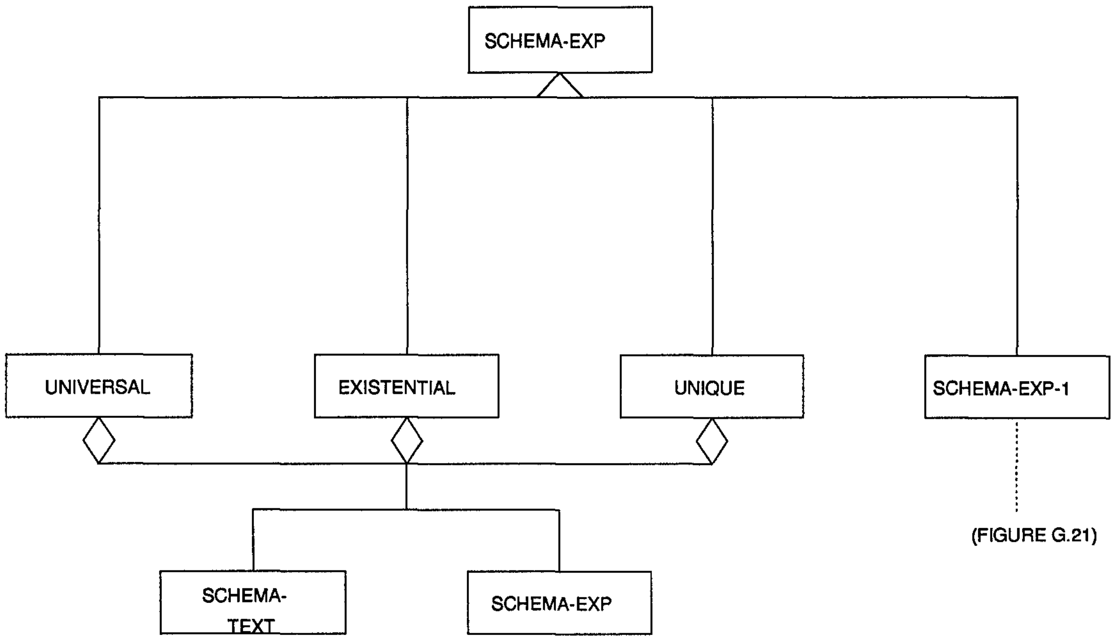

Figure G.20 Schema Calculus Expressions (1 of 4)

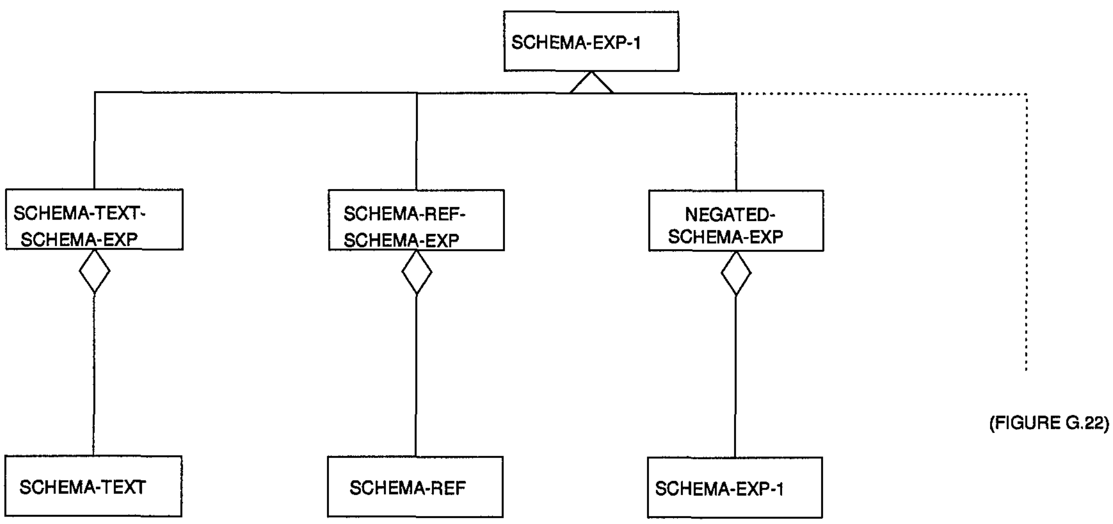

Figure G.21 Schema Calculus Expressions (2 of 4)

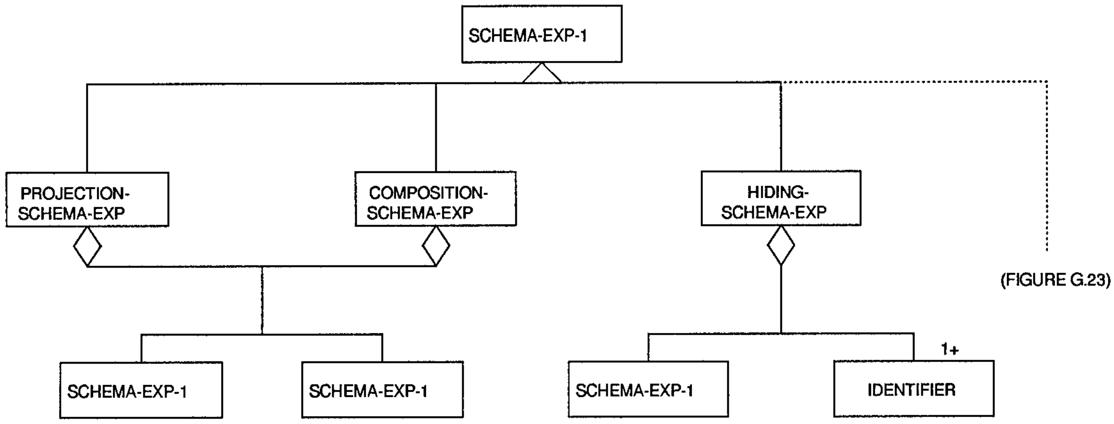

Figure G.22 Schema Calculus Expressions (3 of 4)

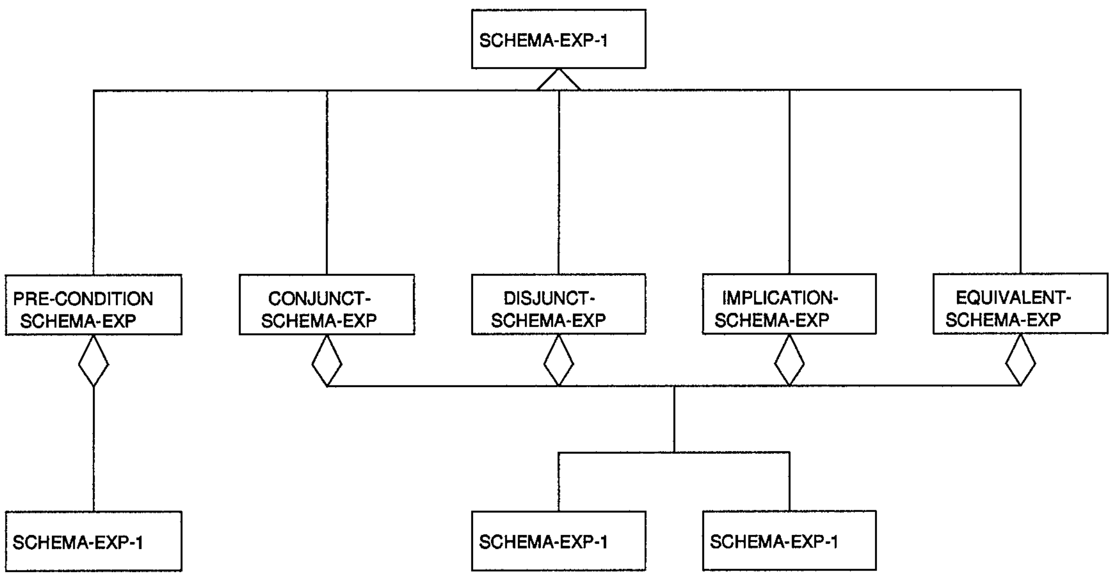

Figure G.23 Schema Calculus Expressions (4 of 4)

G.2 Unified ToolKit

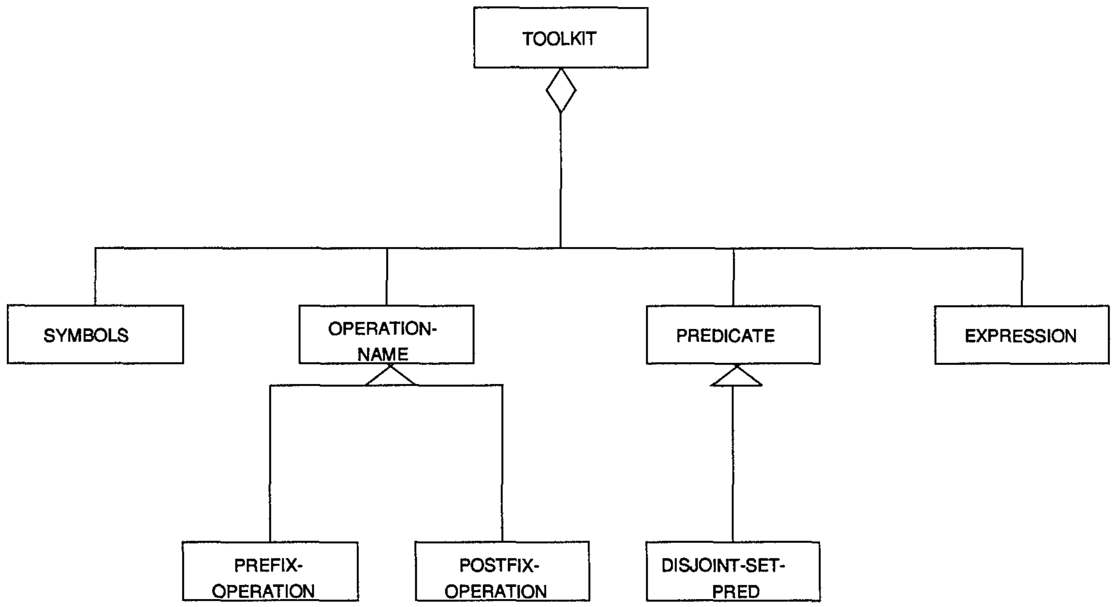

Figure G.24 ToolKit Domain Model

Appendix H. REFINE Code for the UZed Domain Model

This appendix contains the REFINE code for the UZed domain model plus ToolKit. The original Z domain model was revised to use the core unified domain model object classes and map attributes. The UZed extensions include Z specific syntax that are specializations of a unified parent object class found in the Unified Domain Model.

H.1 UZed Domain Model

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: uzed-dm.re

Description:  This file defines an object model for the UZed language and
constructs an AST for an input specification written in LaTeX.

||#

%------------------------------------------------------------------------
% Unified Domain Model Object Class Declarations
%------------------------------------------------------------------------
var Unified-Object              : object-class subtype-of user-object

var DomainTheory                : object-class subtype-of Unified-Object

var DomainTheoryTypes           : object-class subtype-of Unified-Object
  var ObjectTheory                : object-class subtype-of DomainTheoryTypes
  var DynamicTheory               : object-class subtype-of DomainTheoryTypes
  var FunctionalTheory            : object-class subtype-of DomainTheoryTypes

var TheoryId                    : object-class subtype-of Unified-Object
  var ObjectTheoryId              : object-class subtype-of TheoryId
  var DynamicTheoryId             : object-class subtype-of TheoryId
  var FunctionalTheoryId          : object-class subtype-of TheoryId

var TheoryBody                  : object-class subtype-of Unified-Object
  var ObjectTheoryBody            : object-class subtype-of TheoryBody
  var DynamicTheoryBody           : object-class subtype-of TheoryBody
  var FunctionalTheoryBody        : object-class subtype-of TheoryBody

var TheoryDecl                  : object-class subtype-of Unified-Object
  var ObjectTheoryDecl            : object-class subtype-of TheoryDecl
  var DynamicTheoryDecl           : object-class subtype-of TheoryDecl
  var FunctionalTheoryDecl        : object-class subtype-of TheoryDecl
```

```
var SignatureDecl              : object-class subtype-of Unified-Object

var ExternalRef                : object-class subtype-of Unified-Object

var TheoryAxioms               : object-class subtype-of Unified-Object
  var ObjectTheoryAxioms         : object-class subtype-of TheoryAxioms
  var DynamicTheoryAxioms        : object-class subtype-of TheoryAxioms
  var FunctionalTheoryAxioms     : object-class subtype-of TheoryAxioms

%-------------------------------------------------------------------------
% Zed Specific Object Class Declarations
%-------------------------------------------------------------------------
    var StateTheory                : object-class subtype-of DynamicTheory
    var EventTheory                : object-class subtype-of DynamicTheory

  var DefinitionTheory           : object-class subtype-of DomainTheoryTypes
    var BasicTypesDefinition       : object-class subtype-of DefinitionTheory
    var DataTypesDefinition        : object-class subtype-of DefinitionTheory
    var AxiomaticDefinition        : object-class subtype-of DefinitionTheory
    var GenericDefinition          : object-class subtype-of DefinitionTheory
    var SchemaExpressions          : object-class subtype-of DefinitionTheory
    var GlobalConstants            : object-class subtype-of DefinitionTheory

    var StateTheoryBody            : object-class subtype-of DynamicTheoryBody
    var EventTheoryBody            : object-class subtype-of DynamicTheoryBody

  var ZedEnviron                 : object-class subtype-of TheoryBody
  var SyntaxEnviron              : object-class subtype-of TheoryBody
  var AxiomaticBox               : object-class subtype-of TheoryBody
  var GenericBox                 : object-class subtype-of TheoryBody
  var SchemaExpDef               : object-class subtype-of TheoryBody
  var ConstantConstruct          : object-class subtype-of TheoryBody

  var AxiomaticDefDecl           : object-class subtype-of TheoryDecl
  var AxiomaticDefAxioms         : object-class subtype-of TheoryAxioms

  var GenericDefDecl             : object-class subtype-of TheoryDecl
  var GenericDefAxioms           : object-class subtype-of TheoryAxioms

  var StateTheoryDecl            : object-class subtype-of DynamicTheoryDecl
  var StateTheoryAxioms          : object-class subtype-of DynamicTheoryAxioms

  var EventTheoryDecl            : object-class subtype-of DynamicTheoryDecl
    var EventTheoryDecl1           : object-class subtype-of EventTheoryDecl
    var EventTheoryDecl2           : object-class subtype-of EventTheoryDecl
  var EventTheoryAxioms          : object-class subtype-of DynamicTheoryAxioms

var Def-Lhs                    : object-class subtype-of Unified-Object
  var Def-Lhs1                   : object-class subtype-of Def-Lhs

var SchemaText                 : object-class subtype-of Unified-Object
```

```
  var SchemaRef                  : object-class subtype-of ExternalRef
  var SchemaRef1                 : object-class subtype-of ExternalRef
    var UndecoratedSchema          : object-class subtype-of SchemaRef1
    var DecoratedSchema            : object-class subtype-of SchemaRef1
  var SchemaRef2                 : object-class subtype-of ExternalRef
    var DeltaSchema               : object-class subtype-of SchemaRef2
    var XiSchema                  : object-class subtype-of SchemaRef2

  var BasicDeclSeq               : object-class subtype-of SignatureDecl

var Identifier                 : object-class subtype-of Unified-Object

var OperationName              : object-class subtype-of Unified-Object
  var InFixOperation             : object-class subtype-of OperationName
  var ImageOperation             : object-class subtype-of OperationName

var IdName                     : object-class subtype-of Unified-Object

var Num1                       : object-class subtype-of Unified-Object
var Num2                       : object-class subtype-of Unified-Object

var Decoration                 : object-class subtype-of Unified-Object
  var Final-Decoration           : object-class subtype-of Decoration
  var Input-Decoration           : object-class subtype-of Decoration
  var Output-Decoration          : object-class subtype-of Decoration

var MarkerSymbol               : object-class subtype-of Unified-Object

var Symbols                    : object-class subtype-of Unified-Object
  var MiscSymbols                : object-class subtype-of Symbols
  var InFixSymbols               : object-class subtype-of Symbols
    var InRelSymbols              : object-class subtype-of InFixSymbols
      var Equality                 :object-class subtype-of InRelSymbols
      var ElementOf                :object-class subtype-of InRelSymbols
    var IngenSymbols              : object-class subtype-of InFixSymbols
    var InFunSymbols              : object-class subtype-of InFixSymbols
  var PreFixSymbols              : object-class subtype-of Symbols
    var PreRelSymbols             : object-class subtype-of PreFixSymbols
    var PreGenSymbols             : object-class subtype-of PreFixSymbols
  var PostFixSymbols             : object-class subtype-of Symbols
    var PostFunSymbols            : object-class subtype-of PostFixSymbols

var SchemaExp                  : object-class subtype-of Unified-Object
  var Universal-SchemaExp        : object-class subtype-of SchemaExp
  var Existential-SchemaExp      : object-class subtype-of SchemaExp
  var Unique-SchemaExp           : object-class subtype-of SchemaExp
  var SchemaExp-1                : object-class subtype-of SchemaExp
    var SchemaText-SchemaExp      : object-class subtype-of SchemaExp-1
    var SchemaRef-SchemaExp       : object-class subtype-of SchemaExp-1
    var Negated-SchemaExp         : object-class subtype-of SchemaExp-1
```

```
    var PreCondition-SchemaExp      : object-class subtype-of SchemaExp-1
    var Conjunct-SchemaExp          : object-class subtype-of SchemaExp-1
    var Disjunct-SchemaExp          : object-class subtype-of SchemaExp-1
    var Implication-SchemaExp       : object-class subtype-of SchemaExp-1
    var Equivalent-SchemaExp        : object-class subtype-of SchemaExp-1
    var Projection-SchemaExp        : object-class subtype-of SchemaExp-1
    var Hiding-SchemaExp            : object-class subtype-of SchemaExp-1
    var Composition-SchemaExp       : object-class subtype-of SchemaExp-1

var Predicate                   : object-class subtype-of Unified-Object
  var Universal-Pred              : object-class subtype-of Predicate
  var Existential-Pred            : object-class subtype-of Predicate
  var Unique-Pred                 : object-class subtype-of Predicate
  var Predicate-1                 : object-class subtype-of Predicate
    var True-Pred                     : object-class subtype-of Predicate-1
    var False-Pred                    : object-class subtype-of Predicate-1
    var Connective-Pred               : object-class subtype-of Predicate-1
      var Negated-Pred                  : object-class subtype-of Connective-Pred
      var Conjunct-Pred                 : object-class subtype-of Connective-Pred
      var Disjunct-Pred                 : object-class subtype-of Connective-Pred
      var Implication-Pred              : object-class subtype-of Connective-Pred
      var Equivalent-Pred               : object-class subtype-of Connective-Pred
    var Relational1-Pred              : object-class subtype-of Predicate-1
    var Relational2-Pred              : object-class subtype-of Predicate-1
    var Relational3-Pred              : object-class subtype-of Predicate-1
    var PreRel-Pred                   : object-class subtype-of Predicate-1
    var SchemaRef-Pred                : object-class subtype-of Predicate-1
    var PreSchemaRef-Pred             : object-class subtype-of Predicate-1
    var Bracket-Pred                  : object-class subtype-of Predicate-1

var Expression-0                : object-class subtype-of Unified-Object
  var Lambda-Expr                 : object-class subtype-of Expression-0
  var Mu-Expr                     : object-class subtype-of Expression-0
  var Expression                  : object-class subtype-of Expression-0
    var InGen-Expr                  : object-class subtype-of Expression
    var CartesianProd-Expr          : object-class subtype-of Expression
    var Typed-Expr                  : object-class subtype-of Expression
      var Natural-Type                : object-class subtype-of Typed-Expr
      var Positive-Type               : object-class subtype-of Typed-Expr
      var Integer-Type                : object-class subtype-of Typed-Expr
      var Real-Type                   : object-class subtype-of Typed-Expr
      var Character-Type              : object-class subtype-of Typed-Expr
    var Expression-1                : object-class subtype-of Expression
      var InFun-Expr                  : object-class subtype-of Expression-1
      var PowerSet-Expr               : object-class subtype-of Expression-1
      var PreGen-Expr                 : object-class subtype-of Expression-1
      var Negative-Expr               : object-class subtype-of Expression-1
      var PostFun-Expr                : object-class subtype-of Expression-1
      var Exponent-Expr               : object-class subtype-of Expression-1
      var RelImage-Expr               : object-class subtype-of Expression-1
      var Expression-2                : object-class subtype-of Expression-1
```

```
    var FunctionApp-Expr          : object-class subtype-of Expression-2
    var Expression-3              : object-class subtype-of Expression-2
      var Var-Name-Expr             : object-class subtype-of Expression-3
      var Op-Name-Expr              : object-class subtype-of Expression-3
      var Integer-Expr              : object-class subtype-of Expression-3
      var Real-Expr                 : object-class subtype-of Expression-3
      var SchemaRef-Expr            : object-class subtype-of Expression-3
      var Set-Expression            : object-class subtype-of Expression-3
        var Set-Display-Expr          : object-class subtype-of Set-Expression
        var Set-Comp-Expr             : object-class subtype-of Set-Expression
      var Seq-Expr                  : object-class subtype-of Expression-3
      var Bag-Expr                  : object-class subtype-of Expression-3
      var Tuple-Expr                : object-class subtype-of Expression-3
      var Theta-Expr                : object-class subtype-of Expression-3
      var Component-Expr            : object-class subtype-of Expression-3

%-------------------------------------------------------------------------
% Unified Model Attribute Declarations for Branches in Tree Structure
%-------------------------------------------------------------------------
%
% The branches of the Abstract Syntax Tree represent the constructs from the
% right side of the production rules found in the grammar.  These branches are
% annotated with the name of the following REFINE attribute maps.
%
%-------------------------------------------------------------------------
var theory-types        : map(DomainTheory, set(DomainTheoryTypes))
                              computed-using theory-types(x)          = {}

var theory-id           : map(Unified-Object, IdName)                 = {||}
var ot-id               : map(DomainTheoryTypes, IdName)              = {||}
var dt-id               : map(DomainTheoryTypes, IdName)              = {||}
var ft-id               : map(DomainTheoryTypes, IdName)              = {||}

var theory-body         : map(DomainTheoryTypes, TheoryBody)          = {||}
var ot-body             : map(DomainTheoryTypes, ObjectTheoryBody)    = {||}
var dt-body             : map(DomainTheoryTypes, DynamicTheoryBody)   = {||}
var ft-body             : map(DomainTheoryTypes, FunctionalTheoryBody) = {||}

var theory-decl         : map(TheoryBody, TheoryDecl)                 = {||}

var external-refs       : map(TheoryDecl, set(ExternalRef))
                              computed-using external-refs(x)         = {}
var signature-decl      : map(TheoryDecl, set(SignatureDecl))
                              computed-using signature-decl(x)        = {}

var theory-axioms       : map(TheoryBody, TheoryAxioms)               = {||}

%-------------------------------------------------------------------------
% Zed Specific Map Attributes
%-------------------------------------------------------------------------
var st-id               : map(DomainTheoryTypes, IdName)              = {||}
```

```
var et-id                  : map(DomainTheoryTypes, IdName)                 = {||}

var st-body                : map(DomainTheoryTypes, StateTheoryBody)        = {||}
var et-body                : map(DomainTheoryTypes, EventTheoryBody)        = {||}

var zed-environ            : map(Unified-Object, ZedEnviron)                = {||}
var syntax-environ         : map(Unified-Object, SyntaxEnviron)             = {||}
var axiomatic-box          : map(Unified-Object, AxiomaticBox)              = {||}
var generic-box            : map(Unified-Object, GenericBox)                = {||}
var schema-exp-def         : map(Unified-Object, SchemaExpDef)              = {||}
var constant-construct     : map(Unified-Object, ConstantConstruct)         = {||}

var obj-theory-decl        : map(Unified-Object, ObjectTheoryDecl)          = {||}
var obj-theory-axioms      : map(Unified-Object, ObjectTheoryAxioms)        = {||}

var state-theory-decl      : map(Unified-Object, StateTheoryDecl)           = {||}
var state-theory-axioms    : map(Unified-Object, StateTheoryAxioms)         = {||}

var event-theory-decl      : map(Unified-Object, EventTheoryDecl)           = {||}
var event-theory-axioms    : map(Unified-Object, EventTheoryAxioms)         = {||}

var fun-theory-decl        : map(Unified-Object, FunctionalTheoryDecl)      = {||}
var fun-theory-axioms      : map(Unified-Object, FunctionalTheoryAxioms)    = {||}

var axiom-decl-part        : map(Unified-Object, AxiomaticDefDecl)          = {||}
var axiom-axiom-part       : map(Unified-Object, AxiomaticDefAxioms)        = {||}

var generic-decl-part      : map(Unified-Object, GenericDefDecl)            = {||}
var generic-axiom-part     : map(Unified-Object, GenericDefAxioms)          = {||}

var defs-lhs               : map(Unified-Object, Def-Lhs)                   = {||}

var schema-text            : map(Unified-Object, SchemaText)                = {||}
var set-schema-text        : map(Unified-Object, SchemaText)                = {||}

var schema-ref             : map(Unified-Object, SchemaRef)                 = {||}
var schema-inclusion       : map(Unified-Object, set(UndecoratedSchema))
                                 computed-using schema-inclusion(x)         = {}
var schema-inherit         : map(Unified-Object, set(UndecoratedSchema))
                                 computed-using schema-inherit(x)           = {}
var any-schema2-refs       : map(Unified-Object, set(SchemaRef2))
                                 computed-using any-schema2-refs(x)         = {}

var one-event-decl         : map(Unified-Object, BasicDeclSeq)              = {||}
var basic-decls            : map(Unified-Object, set(BasicDeclSeq))
                                   computed-using basic-decls(x)            = {}
var axiom-decls            : map(Unified-Object, set(BasicDeclSeq))
                                   computed-using axiom-decls(x)            = {}
var object-decls           : map(Unified-Object, set(BasicDeclSeq))
                                   computed-using object-decls(x)           = {}
var state-decls            : map(Unified-Object, set(BasicDeclSeq))
```

```
                                          computed-using state-decls(x)          = {}
var event-decls          : map(Unified-Object, set(BasicDeclSeq))
                                          computed-using event-decls(x)          = {}
var functional-decls     : map(Unified-Object, set(BasicDeclSeq))
                                          computed-using functional-decls(x)     = {}

var schema-name          : map(Unified-Object, Identifier)                       = {||}
var var-name             : map(Unified-Object, Identifier)                       = {||}
var def-var-name         : map(Unified-Object, Identifier)                       = {||}
var gen-formals          : map(Unified-Object, set(Identifier))
                                    computed-using gen-formals(x)                = {}
var basic-i-dents        : map(Unified-Object, set(Identifier))
                                    computed-using basic-i-dents(x)              = {}

var any-op-name          : map(Unified-Object, OperationName)                    = {||}

var i-dent               : map(Unified-Object, IdName)                           = {||}
var i-dents              : map(Unified-Object, set(IdName))
                                    computed-using i-dents(x)                    = {}
var enums                : map(Unified-Object, set(IdName))
                                    computed-using enums(x)                      = {}
var para-list            : map(Unified-Object, set(IdName))
                                    computed-using para-list(x)                  = {}
var gen-para-list        : map(Unified-Object, set(IdName))
                                    computed-using gen-para-list(x)              = {}

var num-1                : map(Unified-Object, integer)                          = {||}
var num-2                : map(Unified-Object, real)                             = {||}

var deco-ration          : map(Unified-Object, Decoration)                       = {||}
var any-decoration       : map(Unified-Object, Decoration)                       = {||}
var any-decs             : map(Unified-Object, set(Decoration))
                                    computed-using any-decs(x)                   = {}

var marker1              : map(Unified-Object, MarkerSymbol)                     = {||}
var marker2              : map(Unified-Object, MarkerSymbol)                     = {||}

var any-in-fix-sym       : map(Unified-Object, InFixSymbols)                     = {||}
var any1-in-rel-sym      : map(Unified-Object, InRelSymbols)                     = {||}
var any2-in-rel-sym      : map(Unified-Object, InRelSymbols)                     = {||}
var any3-in-rel-sym      : map(Unified-Object, InRelSymbols)                     = {||}
var any4-in-rel-sym      : map(Unified-Object, InRelSymbols)                     = {||}
var any-pre-rel-sym      : map(Unified-Object, PreRelSymbols)                    = {||}
var any-in-gen-sym       : map(Unified-Object, InGenSymbols)                     = {||}
var any-in-fun-sym       : map(Unified-Object, InFunSymbols)                     = {||}
var any-pre-gen-sym      : map(Unified-Object, PreGenSymbols)                    = {||}
var any-post-fun-sym     : map(Unified-Object, PostFunSymbols)                   = {||}

var schema-exp           : map(Unified-Object, SchemaExp)                        = {||}
var any-schema-exp       : map(Unified-Object, SchemaExp)                        = {||}
var any-schema-exp1      : map(Unified-Object, SchemaExp-1)                      = {||}
```

```
var any1-schema-exp1      : map(Unified-Object, SchemaExp-1)              = {||}
var any2-schema-exp1      : map(Unified-Object, SchemaExp-1)              = {||}

var any-pred              : map(Unified-Object, Predicate)                = {||}
var any-pred1             : map(Unified-Object, Predicate-1)              = {||}
var any1-pred1            : map(Unified-Object, Predicate-1)              = {||}
var any2-pred1            : map(Unified-Object, Predicate-1)              = {||}
var any-preds             : map(Unified-Object, set(Predicate))
                                computed-using any-preds(x)               = {}
var any-axiom-preds       : map(Unified-Object, set(Predicate))
                                computed-using any-axiom-preds(x)         = {}
var any-object-preds      : map(Unified-Object, set(Predicate))
                                computed-using any-object-preds(x)        = {}
var any-state-preds       : map(Unified-Object, set(Predicate))
                                computed-using any-state-preds(x)         = {}
var any-event-preds       : map(Unified-Object, set(Predicate))
                                computed-using any-event-preds(x)         = {}
var any-fun-preds         : map(Unified-Object, set(Predicate))
                                computed-using any-fun-preds(x)           = {}

var any-post-preds        : map(Unified-Object, set(Predicate))
                                computed-using any-post-preds(x)          = {}
var state-post-preds      : map(Unified-Object, set(Predicate))
                                computed-using state-post-preds(x)        = {}
var fun-post-preds        : map(Unified-Object, set(Predicate))
                                computed-using fun-post-preds(x)          = {}

var any-expr0             : map(Unified-Object, Expression-0)             = {||}
var any-expr              : map(Unified-Object, Expression)               = {||}
var any-one-expr          : map(Unified-Object, Expression)               = {||}
var any-second-expr       : map(Unified-Object, Expression)               = {||}
var any-third-expr        : map(Unified-Object, Expression)               = {||}
var any-fourth-expr       : map(Unified-Object, Expression)               = {||}
var any-expr-comp         : map(Unified-Object, Expression)               = {||}
var any-expr1             : map(Unified-Object, Expression-1)             = {||}
var any-one-expr1         : map(Unified-Object, Expression-1)             = {||}
var any-second-expr1      : map(Unified-Object, Expression-1)             = {||}
var any-expr2             : map(Unified-Object, Expression-2)             = {||}
var any-expr3             : map(Unified-Object, Expression-3)             = {||}
var any-exprs             : map(Unified-Object, set(Expression))
                                computed-using any-exprs(x)               = {}
var gen-actual-list       : map(Unified-Object, set(Expression))
                                computed-using gen-actual-list(x)         = {}
var any-exprs1            : map(Unified-Object, set(Expression-1))
                                computed-using any-exprs1(x)              = {}

%------------------------------------------------------------------------
% Annotation Attributes
%------------------------------------------------------------------------
var delta-link            : map(BasicDeclSeq, boolean)                  = {||}
var xi-link               : map(BasicDeclSeq, boolean)                  = {||}
```

```
var inclusion-link       : map(BasicDeclSeq, boolean)                = {||}
var inherit-link         : map(BasicDeclSeq, boolean)                = {||}


%---------------------------------------------------------------------------
% Define Structure of Abstract Syntax Tree
%---------------------------------------------------------------------------
form Define-Tree-Attributes-of-Unified-Specification
  Define-Tree-Attributes('DomainTheory, {'theory-types}) &
  Define-Tree-Attributes('ObjectTheory, {'ot-id, 'ot-body}) &
  Define-Tree-Attributes('DynamicTheory, {'dt-id, 'dt-body}) &
  Define-Tree-Attributes('FunctionalTheory, {'ft-id, 'ft-body}) &

  Define-Tree-Attributes('TheoryBody, {'theory-decl, 'theory-axioms}) &

  Define-Tree-Attributes('TheoryDecl, {'signature-decl, 'external-refs}) &

%---------------------------------------------------------------------------
% Zed Specific AST Structures
%---------------------------------------------------------------------------
  Define-Tree-Attributes('StateTheory, {'st-id, 'st-body}) &
  Define-Tree-Attributes('EventTheory, {'et-id, 'et-body}) &

  Define-Tree-Attributes('BasicTypesDefinition, {'zed-environ}) &
  Define-Tree-Attributes('DataTypesDefinition, {'syntax-environ}) &
  Define-Tree-Attributes('AxiomaticDefinition, {'axiomatic-box}) &
  Define-Tree-Attributes('GenericDefinition, {'generic-box}) &
  Define-Tree-Attributes('SchemaExpressions, {'schema-exp-def}) &
  Define-Tree-Attributes('GlobalConstants, {'constant-construct}) &


  Define-Tree-Attributes('ObjectTheoryBody, {'obj-theory-decl,
                                             'obj-theory-axioms}) &
  Define-Tree-Attributes('StateTheoryBody, {'state-theory-decl,
                                             'state-theory-axioms}) &
  Define-Tree-Attributes('EventTheoryBody, {'event-theory-decl,
                                            'event-theory-axioms}) &
  Define-Tree-Attributes('FunctionalTheoryBody, {'fun-theory-decl,
                                                  'fun-theory-axioms}) &

  Define-Tree-Attributes('ZedEnviron, {'i-dents}) &
  Define-Tree-Attributes('SyntaxEnviron, {'i-dent, 'enums}) &
  Define-Tree-Attributes('AxiomaticBox, {'axiom-decl-part,
                                         'axiom-axiom-part}) &
  Define-Tree-Attributes('GenericBox, {'para-list, 'generic-decl-part,
                                       'generic-axiom-part}) &
  Define-Tree-Attributes('SchemaExpDef, {'theory-id, 'gen-formals,
                                         'schema-exp}) &
  Define-Tree-Attributes('ConstantConstruct, {'defs-lhs, 'any-expr}) &
```

```
Define-Tree-Attributes('AxiomaticDefDecl, {'axiom-decls,
                                           'schema-inclusion}) &
Define-Tree-Attributes('AxiomaticDefAxioms, {'any-axiom-preds}) &

Define-Tree-Attributes('GenericDefDecl, {'basic-decls,
                                         'schema-inclusion}) &
Define-Tree-Attributes('GenericDefAxioms, {'any-preds}) &

Define-Tree-Attributes('ObjectTheoryDecl, {'object-decls,
                                           'schema-inherit}) &
Define-Tree-Attributes('ObjectTheoryAxioms, {'any-object-preds}) &

Define-Tree-Attributes('StateTheoryDecl, {'schema-inclusion,
                                          'state-decls}) &
Define-Tree-Attributes('StateTheoryAxioms, {'any-state-preds,
                                            'state-post-preds}) &

Define-Tree-Attributes('EventTheoryDecl1, {'one-event-decl}) &
Define-Tree-Attributes('EventTheoryDecl2, {'event-decls}) &
Define-Tree-Attributes('EventTheoryAxioms, {'any-event-preds}) &


Define-Tree-Attributes('FunctionalTheoryDecl, {'any-schema2-refs,
                                               'functional-decls}) &
Define-Tree-Attributes('FunctionalTheoryAxioms, {'any-fun-preds,
                                                 'fun-post-preds}) &


Define-Tree-Attributes('Def-Lhs1, {'def-var-name, 'gen-para-list}) &

Define-Tree-Attributes('SchemaText, {'basic-decls, 'any-pred}) &
Define-Tree-Attributes('SchemaRef, {'schema-name}) &

Define-Tree-Attributes('UndecoratedSchema, {'theory-id}) &
Define-Tree-Attributes('DecoratedSchema, {'theory-id, 'any-decs}) &
Define-Tree-Attributes('DeltaSchema, {'theory-id}) &
Define-Tree-Attributes('XiSchema, {'theory-id}) &

Define-Tree-Attributes('BasicDeclSeq, {'basic-i-dents, 'any-expr}) &

Define-Tree-Attributes('Identifier, {'i-dent, 'any-decoration}) &

Define-Tree-Attributes('InFixOperation, {'marker1, 'any-in-fix-sym,
                                         'marker2}) &
Define-Tree-Attributes('ImageOperation, {'marker1, 'marker2}) &

%------------------------------------------------------------------------
% ASTs for Schema Calculus Expressions
%------------------------------------------------------------------------
 Define-Tree-Attributes('Universal-SchemaExp, {'schema-text,
```

```
                                                'any-schema-exp}) &
  Define-Tree-Attributes('Existential-SchemaExp, {'schema-text,
                                                  'any-schema-exp}) &
  Define-Tree-Attributes('Unique-SchemaExp, {'schema-text, 'any-schema-exp}) &
  Define-Tree-Attributes('SchemaText-SchemaExp, {'schema-text}) &
  Define-Tree-Attributes('SchemaRef-SchemaExp, {'schema-ref}) &
  Define-Tree-Attributes('Negated-SchemaExp, {'any-schema-exp1}) &
  Define-Tree-Attributes('PreCondition-SchemaExp, {'any-schema-exp1}) &
  Define-Tree-Attributes('Conjunct-SchemaExp, {'any1-schema-exp1,
                                               'any2-schema-exp1}) &
  Define-Tree-Attributes('Disjunct-SchemaExp, {'any1-schema-exp1,
                                               'any2-schema-exp1}) &
  Define-Tree-Attributes('Implication-SchemaExp, {'any1-schema-exp1,
                                                  'any2-schema-exp1}) &
  Define-Tree-Attributes('Equivalent-SchemaExp, {'any1-schema-exp1,
                                                 'any2-schema-exp1}) &
  Define-Tree-Attributes('Projection-SchemaExp, {'any1-schema-exp1,
                                                 'any2-schema-exp1}) &
  Define-Tree-Attributes('Hiding-SchemaExp, {'any-schema-exp1,
                                             'basic-i-dents}) &
  Define-Tree-Attributes('Composition-SchemaExp, {'any1-schema-exp1,
                                                  'any2-schema-exp1}) &

%-------------------------------------------------------------------------
% ASTs for Predicates
%-------------------------------------------------------------------------
  Define-Tree-Attributes('Universal-Pred, {'schema-text, 'any-pred}) &
  Define-Tree-Attributes('Existential-Pred, {'schema-text, 'any-pred}) &
  Define-Tree-Attributes('Unique-Pred, {'schema-text, 'any-pred}) &
  Define-Tree-Attributes('True-Pred, {'i-dent}) &
  Define-Tree-Attributes('False-Pred, {'i-dent}) &
  Define-Tree-Attributes('Negated-Pred, {'any-pred1}) &
  Define-Tree-Attributes('Conjunct-Pred, {'any1-pred1, 'any2-pred1}) &
  Define-Tree-Attributes('Disjunct-Pred, {'any1-pred1, 'any2-pred1}) &
  Define-Tree-Attributes('Implication-Pred, {'any1-pred1, 'any2-pred1}) &
  Define-Tree-Attributes('Equivalent-Pred, {'any1-pred1, 'any2-pred1}) &
  Define-Tree-Attributes('Relational1-Pred, {'any-one-expr, 'any1-in-rel-sym,
                                             'any-second-expr}) &
  Define-Tree-Attributes('Relational2-Pred, {'any-one-expr, 'any1-in-rel-sym,
                                             'any-second-expr, 'any2-in-rel-sym,
                                             'any-third-expr}) &
  Define-Tree-Attributes('Relational3-Pred, {'any-one-expr, 'any1-in-rel-sym,
                                             'any-second-expr, 'any2-in-rel-sym,
                                             'any-third-expr, 'any3-in-rel-sym,
                                             'any-fourth-expr}) &
  Define-Tree-Attributes('PreRel-Pred, {'any-pre-rel-sym, 'any-expr}) &
  Define-Tree-Attributes('SchemaRef-Pred, {'schema-ref}) &
  Define-Tree-Attributes('PreSchemaRef-Pred, {'schema-ref}) &
  Define-Tree-Attributes('Bracket-Pred, {'any-pred}) &

%-------------------------------------------------------------------------
```

```
% ASTs for Expressions
%-----------------------------------------------------------------------
  Define-Tree-Attributes('Lambda-Expr, {'schema-text, 'any-expr}) &
  Define-Tree-Attributes('Mu-Expr, {'schema-text, 'any-expr}) &
  Define-Tree-Attributes('InGen-Expr, {'any-one-expr, 'any-in-gen-sym,
                                       'any-second-expr}) &
  Define-Tree-Attributes('CartesianProd-Expr, {'any-exprs1}) &
  Define-Tree-Attributes('InFun-Expr, {'any-one-expr1, 'any-in-fun-sym,
                                       'any-second-expr1}) &
  Define-Tree-Attributes('PowerSet-Expr, {'any-expr3}) &
  Define-Tree-Attributes('PreGen-Expr, {'any-pre-gen-sym, 'any-expr3}) &
  Define-Tree-Attributes('Negative-Expr, {'any-expr3}) &
  Define-Tree-Attributes('PostFun-Expr, {'any-expr3, 'any-post-fun-sym}) &
  Define-Tree-Attributes('Exponent-Expr, {'any-expr3, 'any-expr}) &
  Define-Tree-Attributes('RelImage-Expr, {'any-expr3, 'any-expr0}) &
  Define-Tree-Attributes('FunctionApp-Expr, {'any-expr2, 'any-expr3}) &
  Define-Tree-Attributes('Var-Name-Expr, {'var-name, 'gen-actual-list}) &
  Define-Tree-Attributes('Op-Name-Expr, {'any-op-name}) &
  Define-Tree-Attributes('Integer-Expr, {'num-1}) &
  Define-Tree-Attributes('Real-Expr, {'num-2}) &
  Define-Tree-Attributes('Set-Display-Expr, {'any-exprs}) &
  Define-Tree-Attributes('Set-Comp-Expr, {'set-schema-text, 'any-expr}) &
  Define-Tree-Attributes('Seq-Expr, {'any-exprs}) &
  Define-Tree-Attributes('Bag-Expr, {'any-exprs}) &
  Define-Tree-Attributes('Tuple-Expr, {'any-exprs}) &
  Define-Tree-Attributes('Theta-Expr, {'theory-id, 'deco-ration}) &
  Define-Tree-Attributes('Component-Expr, {'any-expr3, 'i-dent})
```

H.2 UToolKit Domain Model

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: utoolkit-dm.re

Description: This file defines an object model for the Unified Mathematical
ToolKit used in conjunction with the UZed language. This ToolKit contains
mathematically simplistic data types which can be used to describe many
different information structures. The ASTs constructed in this program
are automatically appended to the core UZed language tree whenever a ToolKit
element is processed by the parser.
||#

%-------------------------------------------------------------------------
% ToolKit Object Class Declarations
%-------------------------------------------------------------------------

var PreFixOperation             : object-class subtype-of OperationName
var PostFixOperation            : object-class subtype-of OperationName

var SetAlgebraSymbols           : object-class subtype-of MiscSymbols
  var PairFirst                   : object-class subtype-of SetAlgebraSymbols
  var PairSecond                  : object-class subtype-of SetAlgebraSymbols
  var EmptySet                    : object-class subtype-of SetAlgebraSymbols
  var GeneralUnion                : object-class subtype-of SetAlgebraSymbols
  var GeneralIntersect            : object-class subtype-of SetAlgebraSymbols

var RelationSymbols             : object-class subtype-of MiscSymbols
  var RelDomain                   : object-class subtype-of RelationSymbols
  var RelRange                    : object-class subtype-of RelationSymbols

var FunctionSymbols             : object-class subtype-of MiscSymbols

var NumberSymbols               : object-class subtype-of MiscSymbols
  var Successor                   : object-class subtype-of NumberSymbols
  var Cardinality                 : object-class subtype-of NumberSymbols
  var Minimum                     : object-class subtype-of NumberSymbols
  var Maximum                     : object-class subtype-of NumberSymbols

var SequenceSymbols             : object-class subtype-of MiscSymbols
  var HeadSeq                     : object-class subtype-of SequenceSymbols
  var LastSeq                     : object-class subtype-of SequenceSymbols
  var TailSeq                     : object-class subtype-of SequenceSymbols
  var FrontSeq                    : object-class subtype-of SequenceSymbols

var BagSymbols                  : object-class subtype-of MiscSymbols
  var BagCount                    : object-class subtype-of BagSymbols
  var BagItems                    : object-class subtype-of BagSymbols
```

```
var NotEquality          : object-class subtype-of InRelSymbols
var NotElementOf         : object-class subtype-of InRelSymbols
var Z-Subset             : object-class subtype-of InRelSymbols
var PropSubset           : object-class subtype-of InRelSymbols
var LessThan             : object-class subtype-of InRelSymbols
var GreaterThan          : object-class subtype-of InRelSymbols
var LessThanEq           : object-class subtype-of InRelSymbols
var GreaterThanEq        : object-class subtype-of InRelSymbols
var Partition            : object-class subtype-of InRelSymbols
var InBag                : object-class subtype-of InRelSymbols

var DisjointSet          : object-class subtype-of PreRelSymbols

var Relation             : object-class subtype-of InGenSymbols
var Z-Function           : object-class subtype-of InGenSymbols
var PartialFun           : object-class subtype-of InGenSymbols
var Injection            : object-class subtype-of InGenSymbols
var PartialInject        : object-class subtype-of InGenSymbols
var Surjection           : object-class subtype-of InGenSymbols
var PartialSurject       : object-class subtype-of InGenSymbols
var Bijection            : object-class subtype-of InGenSymbols
var FinitePartFun        : object-class subtype-of InGenSymbols
var FinitePartInject     : object-class subtype-of InGenSymbols

var Mapping              : object-class subtype-of InFunSymbols
var NumRange             : object-class subtype-of InFunSymbols
var Addition             : object-class subtype-of InFunSymbols
var Subtraction          : object-class subtype-of InFunSymbols
var SetUnion             : object-class subtype-of InFunSymbols
var SetMinus             : object-class subtype-of InFunSymbols
var Concatenation        : object-class subtype-of InFunSymbols
var BagUnion             : object-class subtype-of InFunSymbols
var Multiplication       : object-class subtype-of InFunSymbols
var Division             : object-class subtype-of InFunSymbols
var Modulo               : object-class subtype-of InFunSymbols
var SetIntersect         : object-class subtype-of InFunSymbols
var Filtering            : object-class subtype-of InFunSymbols
var Composition          : object-class subtype-of InFunSymbols
var BackCompose          : object-class subtype-of InFunSymbols
var OverRiding           : object-class subtype-of InFunSymbols
var DomainRes            : object-class subtype-of InFunSymbols
var RangeRes             : object-class subtype-of InFunSymbols
var AntiDomRes           : object-class subtype-of InFunSymbols
var AntiRangeRes         : object-class subtype-of InFunSymbols

var NonEmpty             : object-class subtype-of PreGenSymbols
var IdRelation           : object-class subtype-of PreGenSymbols
var FiniteSet            : object-class subtype-of PreGenSymbols
var NEFinite             : object-class subtype-of PreGenSymbols
var Sequence             : object-class subtype-of PreGenSymbols
```

```
var NESeq                      : object-class subtype-of PreGenSymbols
var InjSeq                     : object-class subtype-of PreGenSymbols
var Z-Bag                      : object-class subtype-of PreGenSymbols

var Inversion                  : object-class subtype-of PostFunSymbols
var TransClosure               : object-class subtype-of PostFunSymbols
var RefClosure                 : object-class subtype-of PostFunSymbols
%var Iteration                 : object-class subtype-of PostFunSymbols

%----------------------------------------------------------------------
%---------------------Predicate & Expression Objects--------------------
%----------------------------------------------------------------------

var DisjointSet-Pred           : object-class subtype-of PreRel-Pred

var Misc-Expr                  : object-class subtype-of Expression
  var SetAlgebra-Expr            : object-class subtype-of Misc-Expr
    var PairFirst-Expr             : object-class subtype-of SetAlgebra-Expr
    var PairSecond-Expr            : object-class subtype-of SetAlgebra-Expr
    var EmptySet-Expr              : object-class subtype-of SetAlgebra-Expr
    var GeneralUnion-Expr          : object-class subtype-of SetAlgebra-Expr
    var GeneralIntersect-Expr      : object-class subtype-of SetAlgebra-Expr
  var Relations-Expr             : object-class subtype-of Misc-Expr
    var RelDomain-Expr             : object-class subtype-of Relations-Expr
    var RelRange-Expr              : object-class subtype-of Relations-Expr
  var Number-Expr                : object-class subtype-of Misc-Expr
    var Cardinality-Expr           : object-class subtype-of Number-Expr
    var Successor-Expr             : object-class subtype-of Number-Expr
    var MinNumber-Expr             : object-class subtype-of Number-Expr
    var MaxNumber-Expr             : object-class subtype-of Number-Expr

var Relation-Expr              : object-class subtype-of InGen-Expr
var Z-Function-Expr            : object-class subtype-of InGen-Expr
var PartialFun-Expr            : object-class subtype-of InGen-Expr
var Injection-Expr             : object-class subtype-of InGen-Expr
var PartialInject-Expr         : object-class subtype-of InGen-Expr
var Surjection-Expr            : object-class subtype-of InGen-Expr
var PartialSurject-Expr        : object-class subtype-of InGen-Expr
var Bijection-Expr             : object-class subtype-of InGen-Expr
var FinitePartFun-Expr         : object-class subtype-of InGen-Expr
var FinitePartInject-Expr      : object-class subtype-of InGen-Expr

var Mapping-Expr               : object-class subtype-of InFun-Expr
var NumRange-Expr              : object-class subtype-of InFun-Expr
var Addition-Expr              : object-class subtype-of InFun-Expr
var Subtraction-Expr           : object-class subtype-of InFun-Expr
var SetUnion-Expr              : object-class subtype-of InFun-Expr
var SetMinus-Expr              : object-class subtype-of InFun-Expr
var Concatenation-Expr         : object-class subtype-of InFun-Expr
var BagUnion-Expr              : object-class subtype-of InFun-Expr
var Multiplication-Expr        : object-class subtype-of InFun-Expr
```

```
var Division-Expr          : object-class subtype-of InFun-Expr
var Modulo-Expr            : object-class subtype-of InFun-Expr
var SetIntersect-Expr      : object-class subtype-of InFun-Expr
var Filtering-Expr         : object-class subtype-of InFun-Expr
var Composition-Expr       : object-class subtype-of InFun-Expr
var BackCompose-Expr       : object-class subtype-of InFun-Expr
var OverRiding-Expr        : object-class subtype-of InFun-Expr
var DomainRes-Expr         : object-class subtype-of InFun-Expr
var RangeRes-Expr          : object-class subtype-of InFun-Expr
var AntiDomRes-Expr        : object-class subtype-of InFun-Expr
var AntiRangeRes-Expr      : object-class subtype-of InFun-Expr

var  NonEmpty-Expr         : object-class subtype-of PreGen-Expr
var  IdRelation-Expr       : object-class subtype-of PreGen-Expr
var  FiniteSet-Expr        : object-class subtype-of PreGen-Expr
var  NEFinite-Expr         : object-class subtype-of PreGen-Expr
var  Sequence-Expr         : object-class subtype-of PreGen-Expr
var  NESeq-Expr            : object-class subtype-of PreGen-Expr
var  InjSeq-Expr           : object-class subtype-of PreGen-Expr
var  Z-Bag-Expr            : object-class subtype-of PreGen-Expr

var Inversion-Expr         : object-class subtype-of PostFun-Expr
var TransClosure-Expr      : object-class subtype-of PostFun-Expr
var RefClosure-Expr        : object-class subtype-of PostFun-Expr

%------------------------------------------------------------------------
% ToolKit Attribute Declarations for Branches in Tree Structure
%------------------------------------------------------------------------

var any-misc-sym       : map(Unified-Object, MiscSymbols)      = {||}
var any-pre-fix-sym    : map(Unified-Object, PreFixSymbols)    = {||}
var any-post-fix-sym   : map(Unified-Object, PostFixSymbols)   = {||}


%------------------------------------------------------------------------
% Define Structure of ToolKit Abstract Syntax Tree
%------------------------------------------------------------------------
  form Define-Tree-Attributes-of-Zed-Specification

  Define-Tree-Attributes('PreFixOperation, {'any-pre-fix-sym,
                                             'marker1}) &
  Define-Tree-Attributes('PostFixOperation, {'marker1,
                                              'any-post-fix-sym}) &

%------------------------------------------------------------------------
% ASTs for Predicates
%------------------------------------------------------------------------
  Define-Tree-Attributes('DisjointSet-Pred, {'any-expr}) &

%-------------------------------------------------------------------------
% ASTs for Expressions
```

```
%-----------------------------------------------------------------------------
  Define-Tree-Attributes('RelDomain-Expr, {'any-expr}) &
  Define-Tree-Attributes('RelRange-Expr, {'any-expr}) &
  Define-Tree-Attributes('Cardinality-Expr, {'any-expr}) &

  Define-Tree-Attributes('Relation-Expr, {'any-one-expr,
                                         'any-second-expr}) &
  Define-Tree-Attributes('Z-Function-Expr, {'any-one-expr,
                                           'any-second-expr}) &
  Define-Tree-Attributes('PartialFun-Expr, {'any-one-expr,
                                           'any-second-expr}) &
  Define-Tree-Attributes('Injection-Expr, {'any-one-expr,
                                          'any-second-expr}) &
  Define-Tree-Attributes('PartialInject-Expr, {'any-one-expr,
                                              'any-second-expr}) &
  Define-Tree-Attributes('Surjection-Expr, {'any-one-expr,
                                           'any-second-expr}) &
  Define-Tree-Attributes('PartialSurject-Expr, {'any-one-expr,
                                               'any-second-expr}) &
  Define-Tree-Attributes('Bijection-Expr, {'any-one-expr,
                                          'any-second-expr}) &
  Define-Tree-Attributes('FinitePartFun-Expr, {'any-one-expr,
                                              'any-second-expr}) &
  Define-Tree-Attributes('FinitePartInject-Expr, {'any-one-expr,
                                                 'any-second-expr}) &

  Define-Tree-Attributes('Mapping-Expr, {'any-one-expr1,
                                        'any-second-expr1}) &
  Define-Tree-Attributes('NumRange-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('Addition-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('Subtraction-Expr, {'any-one-expr1,
                                            'any-second-expr1}) &
  Define-Tree-Attributes('SetUnion-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('SetMinus-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('Concatenation-Expr, {'any-one-expr1,
                                              'any-second-expr1}) &
  Define-Tree-Attributes('BagUnion-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('Multiplication-Expr, {'any-one-expr1,
                                               'any-second-expr1}) &
  Define-Tree-Attributes('Division-Expr, {'any-one-expr1,
                                         'any-second-expr1}) &
  Define-Tree-Attributes('Modulo-Expr, {'any-one-expr1,
                                       'any-second-expr1}) &
  Define-Tree-Attributes('SetIntersect-Expr, {'any-one-expr1,
                                             'any-second-expr1}) &
  Define-Tree-Attributes('Filtering-Expr, {'any-one-expr1,
```

```
                                                'any-second-expr1}) &
Define-Tree-Attributes('Composition-Expr, {'any-one-expr1,
                                                  'any-second-expr1}) &
Define-Tree-Attributes('BackCompose-Expr, {'any-one-expr1,
                                                  'any-second-expr1}) &
Define-Tree-Attributes('OverRiding-Expr, {'any-one-expr1,
                                                 'any-second-expr1}) &
Define-Tree-Attributes('DomainRes-Expr, {'any-one-expr1,
                                                'any-second-expr1}) &
Define-Tree-Attributes('RangeRes-Expr, {'any-one-expr1,
                                               'any-second-expr1}) &
Define-Tree-Attributes('AntiDomRes-Expr, {'any-one-expr1,
                                                 'any-second-expr1}) &
Define-Tree-Attributes('AntiRangeres-Expr, {'any-one-expr1,
                                                   'any-second-expr1}) &

Define-Tree-Attributes('NonEmpty-Expr, {'any-expr3}) &
Define-Tree-Attributes('IdRelation-Expr, {'any-expr3}) &
Define-Tree-Attributes('FiniteSet-Expr, {'any-expr3}) &
Define-Tree-Attributes('NEFinite-Expr, {'any-expr3}) &
Define-Tree-Attributes('Sequence-Expr, {'any-expr3}) &
Define-Tree-Attributes('NESeq-Expr, {'any-expr3}) &
Define-Tree-Attributes('InjSeq-Expr, {'any-expr3}) &
Define-Tree-Attributes('Z-Bag-Expr, {'any-expr3}) &

Define-Tree-Attributes('Inversion-Expr, {'any-expr3}) &
Define-Tree-Attributes('TransClosure-Expr, {'any-expr3}) &
Define-Tree-Attributes('RefClosure-Expr, {'any-expr3})
```

Appendix I. REFINE *Code for the UZed Grammar*

This appendix contains the REFINE code for the U*Z*ed grammar. The grammar uses the unified domain model and the U*Z*ed extensions (found in Appendices E and H) to construct its production rules. This grammar uses the same common core objects and attributes as the ULARCH grammar to promote the notion of language inheritance. The production rules demonstrate DIALECT'S use of object classes and attributes to form grammar rules. This approach is more intuitive, thereby simplifying a developer's task for forming tokens and creating a hierarchical structure for a formal language.

I.1 UZed Grammar

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: uzed-gram.re

Description:
The following specification defines the Core Z Grammar.  The grammar
uses the unified domain model and the UZed extensions to construct
its production rules.  The grammar incorporates LaTeX specific notations.
||#

%-----------------------------------------------------------------------------
%---------------------------Uzed Language Grammar----------------------------
%-----------------------------------------------------------------------------


!! in-grammar('syntax)

grammar Uzed

 start-classes DomainTheory
 file-classes DomainTheory
 no-patterns
 comments "%" matching "
"
 case-sensitive
 symbol-continue-chars
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_{}[]-|\+-=<>"
 symbol-start-chars
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_\<>"
```

```
productions
 DomainTheory             ::=   ["\\documentstyle[fullpage,zed]{article}"
                                 "\\begin{document}" theory-types + "\\\\"
                                 "\\end{document}"] builds DomainTheory,

 ObjectTheory             ::=   ["\\begin{schema}{"ot-id"}%ObjectTheory"
                                 ot-body]
                                 builds ObjectTheory,

 StateTheory              ::=   ["\\begin{schema}{"st-id"}%StateTheory"
                                 st-body]
                                 builds StateTheory,

 EventTheory              ::=   ["\\begin{schema}{"et-id"}%EventTheory"
                                 et-body]
                                 builds EventTheory,

 FunctionalTheory         ::=   ["\\begin{schema}{"ft-id"}%FunctionalTheory"
                                 ft-body]
                                 builds FunctionalTheory,


 BasicTypesDefinition     ::=   [zed-environ] builds BasicTypesDefinition,

 DataTypesDefinition      ::=   [syntax-environ] builds DataTypesDefinition,

 AxiomaticDefinition      ::=   [axiomatic-box] builds AxiomaticDefinition,

 GenericDefinition        ::=   [generic-box] builds GenericDefinition,

 SchemaExpressions        ::=   [schema-exp-def] builds SchemaExpressions,

 GlobalConstants          ::=   [constant-construct] builds GlobalConstants,


 ObjectTheoryBody         ::=   [obj-theory-decl {[obj-theory-axioms]}
                                 "\\end{schema}"]
                                 builds ObjectTheoryBody,

 StateTheoryBody          ::=   [state-theory-decl {[state-theory-axioms]}
                                 "\\end{schema}"]
                                 builds StateTheoryBody,

 EventTheoryBody          ::=   [{[event-theory-decl]} event-theory-axioms
                                 "\\end{schema}"]
                                 builds EventTheoryBody,

 FunctionalTheoryBody     ::=   [fun-theory-decl {[fun-theory-axioms]}
                                 "\\end{schema}"]
                                 builds FunctionalTheoryBody,
```

```
ZedEnviron            ::=  ["\\begin{zed}" "[" i-dents + "," "]"
                            "\\end{zed}"]
                            builds ZedEnviron,

SyntaxEnviron         ::=  ["\\begin{syntax}" i-dent {["&"]} "::="
                            {["&"]} enums ++ "|"
                            {["\\also" i-dent "&" "::=" "&" enums ++ "|"]}
                            "\\end{syntax}"]
                            builds SyntaxEnviron,

AxiomaticBox          ::=  ["\\begin{axdef}" axiom-decl-part
                            {[axiom-axiom-part]} "\\end{axdef}"]
                            builds AxiomaticBox,

GenericBox            ::=  ["\\begin{gendef}"
                            {["\\[" para-list + "," "\\]"]}
                            generic-decl-part {[generic-axiom-part]}
                            "\\end{gendef}"]
                            builds GenericBox,

SchemaExpDef          ::=  [theory-id {[gen-formals + ","]}
                            "\\defs" schema-exp]
                            builds SchemaExpDef,

ConstantConstruct     ::=  [defs-lhs "==" any-expr]
                            builds ConstantConstruct,

AxiomaticDefDecl      ::=  [axiom-decls + "\\\\"
                            {["\\also" schema-inclusion + "\\\\"]}]
                            builds AxiomaticDefDecl,

AxiomaticDefAxioms    ::=  ["\\where" any-axiom-preds + "\\\\"]
                            builds AxiomaticDefAxioms,

GenericDefDecl        ::=  [basic-decls + "\\\\"
                            {["\\also" schema-inclusion + "\\\\"]}]
                            builds GenericDefDecl,

GenericDefAxioms      ::=  ["\\where" any-preds + "\\\\"]
                            builds GenericDefAxioms,

ObjectTheoryDecl      ::=  [object-decls + "\\\\"
                            {["\\also" schema-inherit + "\\\\"]}]
                            builds ObjectTheoryDecl,

ObjectTheoryAxioms    ::=  ["\\where" any-object-preds + "\\\\"]
                            builds ObjectTheoryAxioms,

StateTheoryDecl       ::=  [schema-inclusion + "\\\\"
                            {[state-decls + "\\\\"]}]
```

```
                              builds StateTheoryDecl,

StateTheoryAxioms        ::=  ["\\where" any-state-preds + "\\\\"
                              {["\\also" state-post-preds + "\\\\"]}]
                              builds StateTheoryAxioms,

EventTheoryDecl1         ::=  [one-event-decl] builds EventTheoryDecl1,

EventTheoryDecl2         ::=  [event-decls ++ "\\\\"] builds EventTheoryDecl2,

EventTheoryAxioms        ::=  ["\\where" any-event-preds + "\\\\"]
                              builds EventTheoryAxioms,

FunctionalTheoryDecl     ::=  [any-schema2-refs + "\\\\" "\\\\"
                              {[functional-decls + "\\\\"]}]
                              builds FunctionalTheoryDecl,

FunctionalTheoryAxioms   ::=  ["\\where" any-fun-preds + "\\\\"
                              {["\\also" fun-post-preds + "\\\\"]}]
                              builds FunctionalTheoryAxioms,

Def-Lhs1                 ::=  [def-var-name
                              {["\\[" gen-para-list + "," "\\]"]}]
                              builds Def-Lhs1,

SchemaText               ::=  [basic-decls + ";" {[ "|" any-pred]}]
                              builds SchemaText,

SchemaRef                ::=  [schema-name] builds SchemaRef,

UndecoratedSchema        ::=  [theory-id] builds UndecoratedSchema,

DecoratedSchema          ::=  [theory-id any-decs + ""] builds DecoratedSchema,

DeltaSchema              ::=  ["\\Delta" theory-id] builds DeltaSchema,

XiSchema                 ::=  ["\\Xi" theory-id] builds XiSchema,

BasicDeclSeq             ::=  [basic-i-dents + "," ":" any-expr]
                              builds BasicDeclSeq,

Identifier               ::=  [i-dent {any-decoration}] builds Identifier,

InFixOperation           ::=  [marker1 any-in-fix-sym marker2]
                              builds InFixOperation,

ImageOperation           ::=  [marker1 "\\limg" marker2 "\\rimg"]
                              builds ImageOperation,

IdName                   ::=  [name] builds IdName,
```

```
Num1                    ::=   [num-1] builds Num1,

Num2                    ::=   [num-2] builds Num2,

Final-Decoration        ::=   ["'"] builds Final-Decoration,

Input-Decoration        ::=   ["?"] builds Input-Decoration,

Output-Decoration       ::=   ["!"] builds Output-Decoration,

MarkerSymbol            ::=   ["_"] builds MarkerSymbol,

Equality                ::=   ["="] builds Equality,

ElementOf               ::=   ["\\in"] builds ElementOf,


%------------------------------------------------------------------------------
%% Syntax for Schema Calculus Expressions
%------------------------------------------------------------------------------

Universal-SchemaExp     ::=   ["\\forall" schema-text "\\spot" any-schema-exp]
                               builds Universal-SchemaExp,

Existential-SchemaExp   ::=   ["\\exists" schema-text "\\spot" any-schema-exp]
                               builds Existential-SchemaExp,

Unique-SchemaExp        ::=   ["\\exists_1" schema-text "\\spot" any-schema-exp]
                               builds Unique-SchemaExp,

SchemaText-SchemaExp    ::=   ["[" schema-text "]"]
                               builds SchemaText-SchemaExp,

SchemaRef-SchemaExp     ::=   [schema-ref]
                               builds SchemaRef-SchemaExp,

Negated-SchemaExp       ::=   ["\\lnot" any-schema-exp1]
                               builds Negated-SchemaExp,

PreCondition-SchemaExp  ::=   ["\\pre" any-schema-exp1]
                               builds PreCondition-SchemaExp,

Conjunct-SchemaExp      ::=   ["(" any1-schema-exp1 "\\land"
                               any2-schema-exp1 ")"]
                               builds Conjunct-SchemaExp,

Disjunct-SchemaExp      ::=   ["(" any1-schema-exp1 "\\lor"
                               any2-schema-exp1 ")"]
                               builds Disjunct-SchemaExp,

Implication-SchemaExp   ::=   ["(" any1-schema-exp1 "\\implies"
```

```
                                  any2-schema-exp1 ")"]
                                  builds Implication-SchemaExp,

Equivalent-SchemaExp     ::=     ["(" any1-schema-exp1 "\\iff"
                                  any2-schema-exp1 ")"]
                                  builds Equivalent-SchemaExp,

Projection-SchemaExp     ::=     ["(" any1-schema-exp1 "\\project"
                                  any2-schema-exp1 ")"]
                                  builds Projection-SchemaExp,

Hiding-SchemaExp         ::=     ["(" any-schema-exp1 "\\hide"
                                  "(" basic-i-dents + "," ")" ")"]
                                  builds Hiding-SchemaExp,

Composition-SchemaExp    ::=     ["(" any1-schema-exp1 "\\semi"
                                  any2-schema-exp1 ")"]
                                  builds Composition-SchemaExp,


%-------------------------------------------------------------------------------
%% Syntax for Predicates
%-------------------------------------------------------------------------------

Universal-Pred           ::=     ["\\forall" schema-text "\\spot" any-pred]
                                  builds Universal-Pred,

Existential-Pred         ::=     ["\\exists" schema-text "\\spot" any-pred]
                                  builds Existential-Pred,

Unique-Pred              ::=     ["\\exists_1" schema-text "\\spot" any-pred]
                                  builds Unique-Pred,

True-Pred                ::=     ["True"] builds True-Pred,

False-Pred               ::=     ["False"] builds False-Pred,

Negated-Pred             ::=     ["\\lnot" any-pred1] builds Negated-Pred,

Conjunct-Pred            ::=     ["(" any1-pred1 "\\land" any2-pred1 ")"]
                                  builds Conjunct-Pred,

Disjunct-Pred            ::=     ["(" any1-pred1 "\\lor" any2-pred1 ")"]
                                  builds Disjunct-Pred,

Implication-Pred         ::=     ["(" any1-pred1 "\\implies" any2-pred1 ")"]
                                  builds Implication-Pred,

Equivalent-Pred          ::=     ["(" any1-pred1 "\\iff" any2-pred1 ")"]
                                  builds Equivalent-Pred,
```

```
Relational1-Pred          ::=  [any-one-expr any1-in-rel-sym any-second-expr]
                                builds Relational1-Pred,

Relational2-Pred          ::=  [any-one-expr any1-in-rel-sym any-second-expr
                                any2-in-rel-sym any-third-expr]
                                builds Relational2-Pred,

Relational3-Pred          ::=  [any-one-expr any1-in-rel-sym any-second-expr
                                any2-in-rel-sym any-third-expr
                                any3-in-rel-sym any-fourth-expr]
                                builds Relational3-Pred,

SchemaRef-Pred            ::=  [schema-ref] builds SchemaRef-Pred,

PreSchemaRef-Pred         ::=  ["\\pre" schema-ref]
                                builds PreSchemaRef-Pred,

Bracket-Pred              ::=  ["(" any-pred ")"] builds Bracket-Pred,


%-------------------------------------------------------------------------------
%% Syntax for Expressions
%-------------------------------------------------------------------------------

Lambda-Expr               ::=  ["\\lambda" schema-text "\\spot" any-expr]
                                builds Lambda-Expr,

Mu-Expr                   ::=  ["\\mu" schema-text {["\\spot" any-expr]}]
                                builds Mu-Expr,

CartesianProd-Expr        ::=  [any-exprs1 ++ "\\cross"]
                                builds CartesianProd-Expr,

Natural-Type              ::=  ["\\nat"] builds Natural-Type,
Positive-Type             ::=  ["\\nat_1"] builds Positive-Type,
Integer-Type              ::=  ["\\num"] builds Integer-Type,
Real-Type                 ::=  ["\\cal R"] builds Real-Type,
Character-Type            ::=  ["\\seq CHAR"] builds Character-Type,

PowerSet-Expr             ::=  ["\\power" any-expr3] builds PowerSet-Expr,

Negative-Expr             ::=  ["-" any-expr3] builds Negative-Expr,

Exponent-Expr             ::=  [any-expr3 "\\bsup" any-expr "\\esup"]
                                builds Exponent-Expr,

RelImage-Expr             ::=  [any-expr3 "\\limg" any-expr0 "\\rimg"]
                                builds RelImage-Expr,

FunctionApp-Expr          ::=  [any-expr2 any-expr3] builds FunctionApp-Expr,
```

```
Var-Name-Expr        ::=  [var-name {["\\[" gen-actual-list + "," "\\]"]}]
                           builds Var-Name-Expr,

Op-Name-Expr         ::=  ["(" any-op-name ")"] builds Op-Name-Expr,

Integer-Expr         ::=  [num-1] builds Integer-Expr,
Real-Expr            ::=  [num-2] builds Real-Expr,

Set-Display-Expr     ::=  ["\\{" any-exprs * "," "\\}"]
                           builds Set-Display-Expr,

Set-Comp-Expr        ::=  ["\\{" set-schema-text
                           {["\\spot" any-expr]} "\\}"]
                           builds Set-Comp-Expr,

Seq-Expr             ::=  ["\\langle" any-exprs * "," "\\rangle"]
                           builds Seq-Expr,

Bag-Expr             ::=  ["\\lbag" any-exprs * "," "\\rbag"]
                           builds Bag-Expr,

Tuple-Expr           ::=  ["(" any-exprs ++ "," ")"] builds Tuple-Expr,

Theta-Expr           ::=  ["\\theta" theory-id deco-ration]
                           builds Theta-Expr,

Component-Expr       ::=  [any-expr3 "." i-dent] builds Component-Expr

 precedence

    for schemaexp brackets "(" matching ")"
      (same-level "\\\\", ";" associativity left),
      (same-level "\\semi" associativity left),
      (same-level "\\hide" associativity left),
      (same-level "\\project" associativity left),
      (same-level "\\iff" associativity left),
      (same-level "\\implies" associativity right),
      (same-level "\\lor" associativity left),
      (same-level "\\land" associativity left),
      (same-level "\\pre" associativity none),
      (same-level "\\lnot" associativity none),
      (same-level "=", "\\in" associativity none),
      (same-level "\\cross" associativity left),
      (same-level "\\power" associativity right)
end
```

I.2 UToolKit Grammar

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: utoolkit-gram.re

Description:
The following specification defines the Unified Mathematical ToolKit Grammar.
The grammar uses the unified domain model and the UZed extensions to
construct its production rules.  The grammar incorporates LaTeX specific
notations.
||#


!! in-grammar('syntax)

grammar UToolKit

 inherits-from Uzed
 start-classes DomainTheory

 file-classes DomainTheory
 no-patterns
 comments "%" matching "
"
 case-sensitive
 symbol-continue-chars
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789_{}[]-|\+-=<>"
 symbol-start-chars
 "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_\<>"


 productions

  PreFixOperation          ::=   [any-pre-fix-sym marker1]
                                  builds PreFixOperation,

  PostFixOperation         ::=   [marker1 any-post-fix-sym]
                                  builds PostFixOperation,

%------------------------------------------------------------------------------
%% MiscSymbols
%------------------------------------------------------------------------------

%%% SetAlgebra

  PairFirst              ::=    ["first"] builds PairFirst,
  PairSecond             ::=    ["second"] builds PairSecond,
  EmptySet               ::=    ["\\empty"] builds EmptySet,
```

```
  GeneralUnion          ::=    ["\\bigcup"] builds GeneralUnion,
  GeneralIntersect      ::=    ["\\bigcap"] builds GeneralIntersect,

%%% Relations

  RelDomain             ::=    ["\\dom"] builds RelDomain,
  RelRange              ::=    ["\\ran"] builds RelRange,

%%% Functions

%%% Numbers

  Successor             ::=    ["\\succ"] builds Successor,
  Cardinality           ::=    ["\\#"] builds Cardinality,
  Minimum               ::=    ["\\min"] builds Minimum,
  Maximum               ::=    ["\\max"] builds Maximum,

%%% Sequences

  HeadSeq               ::=    ["head"] builds HeadSeq,
  LastSeq               ::=    ["last"] builds LastSeq,
  TailSeq               ::=    ["tail"] builds TailSeq,
  FrontSeq              ::=    ["front"] builds FrontSeq,

%%% Bags

  BagCount              ::=    ["count"] builds BagCount,
  BagItems              ::=    ["items"] builds BagItems,

%-----------------------------------------------------------------------------
%% InRelSymbols
%-----------------------------------------------------------------------------
  NotEquality           ::=    ["\\neq"] builds NotEquality,
  NotElementOf          ::=    ["\\notin"] builds NotElementOf,
  Z-Subset              ::=    ["\\subset"] builds Z-Subset,
  PropSubset            ::=    ["\\subseteq"] builds PropSubset,
  LessThan              ::=    ["<"] builds LessThan,
  GreaterThan           ::=    [">"] builds GreaterThan,
  LessThanEq            ::=    ["\\leq"] builds LessThanEq,
  GreaterThanEq         ::=    ["\\geq"] builds GreaterThanEq,
  Partition             ::=    ["\\partition"] builds Partition,
  InBag                 ::=    ["\\inbag"] builds InBag,

%-----------------------------------------------------------------------------
%% PreRelSymbols
%-----------------------------------------------------------------------------
  DisjointSet           ::=    ["\\disjoint"] builds DisjointSet,

%-----------------------------------------------------------------------------
%% InGenSymbols
%-----------------------------------------------------------------------------
```

```
  Relation                ::=    ["\\rel"] builds Relation,
  Z-Function              ::=    ["\\fun"] builds Z-Function,
  PartialFun              ::=    ["\\pfun"] builds PartialFun,
  Injection               ::=    ["\\inj"] builds Injection,
  PartialInject           ::=    ["\\pinj"] builds PartialInject,
  Surjection              ::=    ["\\surj"] builds Surjection,
  PartialSurject          ::=    ["\\psurj"] builds PartialSurject,
  Bijection               ::=    ["\\bij"] builds Bijection,
  FinitePartFun           ::=    ["\\ffun"] builds FinitePartFun,
  FinitePartInject        ::=    ["\\finj"] builds FinitePartInject,

%------------------------------------------------------------------------------
%% InFunSymbols
%------------------------------------------------------------------------------
  Mapping                 ::=    ["\\mapsto"] builds Mapping,
  NumRange                ::=    ["\\upto"] builds NumRange,
  Addition                ::=    ["+"] builds Addition,
  Subtraction             ::=    ["-"] builds Subtraction,
  SetUnion                ::=    ["\\cup"] builds SetUnion,
  SetMinus                ::=    ["\\setminus"] builds SetMinus,
  Concatenation           ::=    ["\\cat"] builds Concatenation,
  BagUnion                ::=    ["\\uplus"] builds BagUnion,
  Multiplication          ::=    ["*"] builds Multiplication,
  Division                ::=    ["\\div"] builds Division,
  Modulo                  ::=    ["\\mod"] builds Modulo,
  SetIntersect            ::=    ["\\cap"] builds SetIntersect,
  Filtering               ::=    ["\\filter"] builds Filtering,
  Composition             ::=    ["\\comp"] builds Composition,
  BackCompose             ::=    ["\\circ"] builds BackCompose,
  OverRiding              ::=    ["\\oplus"] builds OverRiding,
  DomainRes               ::=    ["\\dres"] builds DomainRes,
  RangeRes                ::=    ["\\rres"] builds RangeRes,
  AntiDomRes              ::=    ["\\ndres"] builds AntiDomRes,
  AntiRangeRes            ::=    ["\\nrres"] builds AntiRangeRes,

%------------------------------------------------------------------------------
%% PreGenSymbols
%------------------------------------------------------------------------------
  NonEmpty                ::=    ["\\power_1"] builds NonEmpty,
  IdRelation              ::=    ["\\id"] builds IdRelation,
  FiniteSet               ::=    ["\\finset"] builds FiniteSet,
  NEFinite                ::=    ["\\finset_1"] builds NEFinite,
  Sequence                ::=    ["\\seq"] builds Sequence,
  NESeq                   ::=    ["\\seq_1"] builds NESeq,
  InjSeq                  ::=    ["\\iseq"] builds InjSeq,
  Z-Bag                   ::=    ["\\bag"] builds Z-Bag,

%------------------------------------------------------------------------------
%% PostFunSymbols
%------------------------------------------------------------------------------
  Inversion               ::=    ["\\inv"] builds Inversion,
```

```
  TransClosure          ::=    ["\\plus"] builds TransClosure,
  RefClosure            ::=    ["\\star"] builds RefClosure,
%  Iteration            ::=   ["\\bsup" "n" "\\esup"] builds Iteration,

%------------------------------------------------------------------------------
% Syntax for Predicates & Expressions
%------------------------------------------------------------------------------
%% Syntax for PreRel-Predicates

  DisjointSet-Pred      ::=    ["\\disjoint" any-expr]
                                builds DisjointSet-Pred,

%------------------------------------------------------------------------------
% Syntax for Misc-Expressions
%------------------------------------------------------------------------------

  EmptySet-Expr         ::=    ["\\{\\empty\\}"] builds EmptySet-Expr,

  RelDomain-Expr        ::=    ["\\dom" any-expr] builds RelDomain-Expr,

  RelRange-Expr         ::=    ["\\ran" any-expr] builds RelRange-Expr,

  Cardinality-Expr      ::=    ["\\#" any-expr] builds Cardinality-Expr,

%------------------------------------------------------------------------------
%% Syntax for InGen-Expressions
%------------------------------------------------------------------------------
  Relation-Expr         ::=    ["(" any-one-expr "\\rel" any-second-expr ")"]
                                builds Relation-Expr,

  Z-Function-Expr       ::=    ["(" any-one-expr "\\fun" any-second-expr ")"]
                                builds Z-Function-Expr,

  PartialFun-Expr       ::=    ["(" any-one-expr "\\pfun" any-second-expr ")"]
                                builds PartialFun-Expr,

  Injection-Expr        ::=    ["(" any-one-expr "\\inj" any-second-expr ")"]
                                builds Injection-Expr,

  PartialInject-Expr    ::=    ["(" any-one-expr "\\pinj" any-second-expr")" ]
                                builds PartialInject-Expr,

  Surjection-Expr       ::=    ["(" any-one-expr "\\surj" any-second-expr")" ]
                                builds Surjection-Expr,

  PartialSurject-Expr   ::=    ["(" any-one-expr "\\psurj" any-second-expr")" ]
                                builds PartialSurject-Expr,

  Bijection-Expr        ::=    ["(" any-one-expr "\\bij" any-second-expr")" ]
                                builds Bijection-Expr,
```

```
  FinitePartFun-Expr     ::=   ["(" any-one-expr "\\ffun" any-second-expr")" ]
                                builds FinitePartFun-Expr,

  FinitePartInject-Expr  ::=   ["(" any-one-expr "\\finj" any-second-expr")" ]
                                builds FinitePartInject-Expr,

%-----------------------------------------------------------------------
%% Syntax for InFun-Expressions
%-----------------------------------------------------------------------
  Mapping-Expr           ::=   [any-one-expr1 "\\mapsto" any-second-expr1]
                                builds Mapping-Expr,

  NumRange-Expr          ::=   [any-one-expr1 "\\upto" any-second-expr1]
                                builds NumRange-Expr,

  Addition-Expr          ::=   [any-one-expr1 "+" any-second-expr1]
                                builds Addition-Expr,

  Subtraction-Expr       ::=   [any-one-expr1 "-" any-second-expr1]
                                builds Subtraction-Expr,

  SetUnion-Expr          ::=   [any-one-expr1 "\\cup" any-second-expr1]
                                builds SetUnion-Expr,
  SetMinus-Expr          ::=   [any-one-expr1 "\\setminus" any-second-expr1]
                                builds SetMinus-Expr,
  Concatenation-Expr     ::=   [any-one-expr1 "\\cat" any-second-expr1]
                                builds Concatenation-Expr,
  BagUnion-Expr          ::=   [any-one-expr1 "\\uplus" any-second-expr1]
                                builds BagUnion-Expr,
  Multiplication-Expr    ::=   [any-one-expr1 "*" any-second-expr1]
                                builds Multiplication-Expr,
  Division-Expr          ::=   [any-one-expr1 "\\div" any-second-expr1]
                                builds Division-Expr,
  Modulo-Expr            ::=   [any-one-expr1 "\\mod" any-second-expr1]
                                builds Modulo-Expr,
  SetIntersect-Expr      ::=   [any-one-expr1 "\\cap" any-second-expr1]
                                builds SetIntersect-Expr,
  Filtering-Expr         ::=   [any-one-expr1 "\\filter" any-second-expr1]
                                builds Filtering-Expr,
  Composition-Expr       ::=   [any-one-expr1 "\\comp" any-second-expr1]
                                builds Composition-Expr,
  BackCompose-Expr       ::=   [any-one-expr1 "\\circ" any-second-expr1]
                                builds BackCompose-Expr,
  OverRiding-Expr        ::=   [any-one-expr1 "\\oplus" any-second-expr1]
                                builds OverRiding-Expr,
  DomainRes-Expr         ::=   [any-one-expr1 "\\dres" any-second-expr1]
                                builds DomainRes-Expr,
  RangeRes-Expr          ::=   [any-one-expr1 "\\rres" any-second-expr1]
                                builds RangeRes-Expr,
  AntiDomRes-Expr        ::=   [any-one-expr1 "\\ndres" any-second-expr1]
                                builds AntiDomRes-Expr,
```

```
  AntiRangeRes-Expr      ::=     [any-one-expr1 "\\nrres" any-second-expr1]
                                  builds AntiRangeRes-Expr,

%------------------------------------------------------------------------------
%% Syntax for PreGen-Expressions
%------------------------------------------------------------------------------
  NonEmpty-Expr          ::=     ["\\power_1" any-expr3] builds NonEmpty-Expr,
  IdRelation-Expr        ::=     ["\\id" any-expr3] builds IdRelation-Expr,
  FiniteSet-Expr         ::=     ["\\finset" any-expr3] builds FiniteSet-Expr,
  NEFinite-Expr          ::=     ["\\finset_1" any-expr3] builds NEFinite-Expr,
  Sequence-Expr          ::=     ["\\seq" any-expr3] builds Sequence-Expr,
  NESeq-Expr             ::=     ["\\seq_1" any-expr3] builds NESeq-Expr,
  InjSeq-Expr            ::=     ["\\iseq" any-expr3] builds InjSeq-Expr,
  Z-Bag-Expr             ::=     ["\\bag" any-expr3] builds Z-Bag-Expr,

%------------------------------------------------------------------------------
%% Syntax for PostFun-Expressions
%------------------------------------------------------------------------------
  Inversion-Expr         ::=     [any-expr3 "\\inv"] builds Inversion-Expr,
  TransClosure-Expr      ::=     [any-expr3 "\\plus"] builds TransClosure-Expr,
  RefClosure-Expr        ::=     [any-expr3 "\\star"] builds RefClosure-Expr
%  Iteration            ::=    [any-expr3 "\\bsup" "n" "\\esup"] builds Iteration,

 precedence

    for expression brackets "(" matching ")"
      (same-level "\\\\", ";" associativity left),
      (same-level "\\dom" associativity right),
      (same-level "\\#" associativity right),
      (same-level "<", "\\leq", "\\geq", ">", "\\neq", "\\notin", "\\subset",
                  "\\subseteq", "\\partition", "\\inbag" associativity none),
      (same-level "\\mapsto" associativity left),
      (same-level "\\upto" associativity left),
      (same-level "+", "-", "\\cup", "\\setminus", "\\cat", "\\uplus"
                  associativity left),
      (same-level "*", "\\div", "\\mod", "\\cap", "\\comp", "\\circ", "\\filter"
                  associativity left),
      (same-level "\\oplus" associativity left),
      (same-level "\\dres", "\\rres", "\\ndres", "\\nrres" associativity left)

end
```

Appendix J. Semantic Analysis Code

This appendix contains the REFINE code for the U*Z*ed semantic analysis tasks. These tasks augment the ASTs through the expansion of shorthand notations, and complete the intermediate representation prior to execution. There are currently five completed semantic analysis tasks, all detailed in the sections below.

J.1 Schema Inclusion

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: u-schema-inclusion.re

Description:
The following Refine program traverses a UZed AST and then augments the tree
with any included schemas by merging the signatures and conjuncting the predicates of the include
||#

function make-schema-inclusion()  =
  let (Temp-Basic-Seq  : BasicDeclSeq = make-object('BasicDeclSeq))
  let (specs           : set(DomainTheoryTypes) =
                       instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Symbol     : Symbol = newsymbol('TempSymbol))
  let (Temp-Predicate  : Predicate = make-object('Predicate))

 (enumerate zs over specs do
   if(defined? (st-body(zs))) then
     (enumerate i over descendants-of-class(st-body(zs), 'UndecoratedSchema) do
      (enumerate j over descendants-of-class(i, 'IdName) do
       (Temp-Symbol <- name(j);
        enumerate k over specs do
        (if (Temp-Symbol = name(ot-id(k))) then
         (enumerate l over descendants-of-class(ot-body(k), 'BasicDeclSeq) do
  (Temp-Basic-Seq <- l;
           set-attrs((Temp-Basic-Seq), 'inclusion-link, true);
   state-decls(state-theory-decl(st-body(zs))) <-
   state-decls(state-theory-decl(st-body(zs))) with
           Temp-Basic-Seq
           )
          );
         (enumerate m over descendants-of-class(ot-body(k), 'Predicate) do
  (Temp-Predicate <- m;
   state-post-preds(state-theory-axioms(st-body(zs))) <-
           state-post-preds(state-theory-axioms(st-body(zs))) with
   Temp-Predicate
           )
          )
         )
        )
       )
      )
   )

%%
```

J.2 Δ Notation Expansion

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: u-delta-schemas.re

Description:  A Delta reference is expanded by unioning the signature of the
Delta schema with the signature of the referenced schema.  Likewise, the
predicates of both schemas are conjuncted together.  A boolean annotation
attribure, ``delta-link'', is set to true.
||#

function make-delta-schemas () =
  let (Temp-Basic-Seq   : BasicDeclSeq = make-object('BasicDeclSeq))
  let (specs            : set(DomainTheoryTypes) =
                        instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Symbol      : Symbol = newsymbol('TempSymbol))
  let (Temp-Predicate   : Predicate = make-object('Predicate))

 (enumerate zs over specs do
   if(defined? (ft-body(zs))) then
     (enumerate i over descendants-of-class(ft-body(zs), 'DeltaSchema) do
      (enumerate j over descendants-of-class(i, 'IdName) do
       (Temp-Symbol <- name(j);
        enumerate k over specs do
        (if (Temp-Symbol = name(ot-id(k))) then
         (enumerate l over descendants-of-class(ot-body(k), 'BasicDeclSeq) do
  (Temp-Basic-Seq <- l;
          set-attrs((Temp-Basic-Seq), 'delta-link, true);
   functional-decls(fun-theory-decl(ft-body(zs))) <-
   functional-decls(fun-theory-decl(ft-body(zs))) with
          Temp-Basic-Seq
          )
         );
        (enumerate m over descendants-of-class(ot-body(k), 'Predicate) do
  (Temp-Predicate <- m;
   any-fun-preds(fun-theory-axioms(ft-body(zs))) <-
          any-fun-preds(fun-theory-axioms(ft-body(zs))) with
   Temp-Predicate
          )
         )
        )
       )
      )
     )
   )

%%
```

```
!! in-package("RU")
!! in-grammar('user)

#||
 File name: u-delta-vars.re

Description:  The Delta schema's signature is expanded a second time to
include the ticked counterparts of the referenced object's variables.  The
predicate is expanded to include the corresponding constraints placed on those
ticked variables.
||#

function make-delta-vars ()  =
  let (Temp-Identifier  : Identifier = make-object('Identifier))
  let (specs            : set(DomainTheoryTypes) =
                        instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Idents      : set(Identifier) = {})
  let (Temp-Expression  : Expression = make-object('Expression))
  let (Temp-DeclSeq     : BasicDeclSeq = make-object('BasicDeclSeq))


 (enumerate zs over specs do
   if(defined? (ft-body(zs))) then
     (enumerate i over descendants-of-class(ft-body(zs), 'BasicDeclSeq) do
       if(defined? (delta-link(i))) then
         (Temp-Idents <- {};
           enumerate j over descendants-of-class(i, 'Identifier) do
           Temp-Identifier <-
            set-attrs(make-object('Identifier),
                                 'i-dent,
                                   set-attrs(make-object('IdName),
                                              'Name, name(i-dent(j))),
                                 'any-decoration,
                                   (make-object ('Final-Decoration)));
           Temp-Idents <- Temp-Idents with Temp-Identifier
           );
          (enumerate k over descendants-of-class(i, 'Expression) do
           Temp-Expression <- k
           );
           Temp-DeclSeq <- set-attrs(make-object('BasicDeclSeq),
                                     'basic-i-dents, Temp-Idents,
                                     'any-expr, Temp-Expression);
           functional-decls(fun-theory-decl(ft-body(zs))) <-
           functional-decls(fun-theory-decl(ft-body(zs))) with
           Temp-DeclSeq
       )
   )

%%
```

J.3 Ξ Notation Expansion

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: u-xi-schemas.re

Description:  A Xi reference is expanded by unioning the signature of the
Xi schema with the signature of the referenced schema.  Likewise, the
predicates of both schemas are conjuncted together.  A boolean annotation
attribure, ``xi-link'', is set to true.
||#

function make-xi-schemas ()  =
  let (Temp-Basic-Seq   : BasicDeclSeq = make-object('BasicDeclSeq))
  let (specs            : set(DomainTheoryTypes) =
                        instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Symbol      : Symbol = newsymbol('TempSymbol))
  let (Temp-Predicate   : Predicate = make-object('Predicate))

 (enumerate zs over specs do
   if(defined? (ft-body(zs))) then
     (enumerate i over descendants-of-class(ft-body(zs), 'XiSchema) do
      (enumerate j over descendants-of-class(i, 'IdName) do
       (Temp-Symbol <- name(j);
        enumerate k over specs do
        (if (Temp-Symbol = name(ot-id(k))) then
         (enumerate l over descendants-of-class(ot-body(k), 'BasicDeclSeq) do
  (Temp-Basic-Seq <- l;
          set-attrs((Temp-Basic-Seq), 'xi-link, true);
   functional-decls(fun-theory-decl(ft-body(zs))) <-
   functional-decls(fun-theory-decl(ft-body(zs))) with
          Temp-Basic-Seq
          )
         );
        (enumerate m over descendants-of-class(ot-body(k), 'Predicate) do
  (Temp-Predicate <- m;
   any-fun-preds(fun-theory-axioms(ft-body(zs))) <-
          any-fun-preds(fun-theory-axioms(ft-body(zs))) with
   Temp-Predicate
          )
         )
        )
       )
      )
     )
   )

%%
```

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: u-xi-vars.re

Description:  The Xi schema's signature is expanded a second time to
include the ticked counterparts of the referenced object's variables.  The
predicate is expanded to include the corresponding constraints placed on those
ticked variables.
||#

function make-xi-vars ()  =
  let (Temp-Identifier  : Identifier = make-object('Identifier))
  let (specs            : set(DomainTheoryTypes) =
                        instances(find-object-class('DomainTheoryTypes), true))
  let (Temp-Idents      : set(Identifier) = {})
  let (Temp-Expression  : Expression = make-object('Expression))
  let (Temp-DeclSeq     : BasicDeclSeq = make-object('BasicDeclSeq))

 (enumerate zs over specs do
   if(defined? (ft-body(zs))) then
     (enumerate i over descendants-of-class(ft-body(zs), 'BasicDeclSeq) do
       if(defined? (xi-link(i))) then
         (Temp-Idents <- {};
           enumerate j over descendants-of-class(i, 'Identifier) do
           Temp-Identifier <-
            set-attrs(make-object('Identifier),
                                 'i-dent,
                                   set-attrs(make-object('IdName),
                                             'Name, name(i-dent(j))),
                                 'any-decoration,
                                   (make-object ('Final-Decoration)));
           Temp-Idents <- Temp-Idents with Temp-Identifier
           );
          (enumerate k over descendants-of-class(i, 'Expression) do
           Temp-Expression <- k
           );
           Temp-DeclSeq <- set-attrs(make-object('BasicDeclSeq),
                                    'basic-i-dents, Temp-Idents,
                                    'any-expr, Temp-Expression);
           functional-decls(fun-theory-decl(ft-body(zs))) <-
           functional-decls(fun-theory-decl(ft-body(zs))) with
           Temp-DeclSeq
       )
   )

%%
```

Appendix K. Execution Target Domain Model

This appendix contains the object-oriented domain model for the execution framework. REFINE is the target framework, and the following figures identify the principal object classes. This model guides the development of the execution framework mappings providing a concrete target for the source objects.

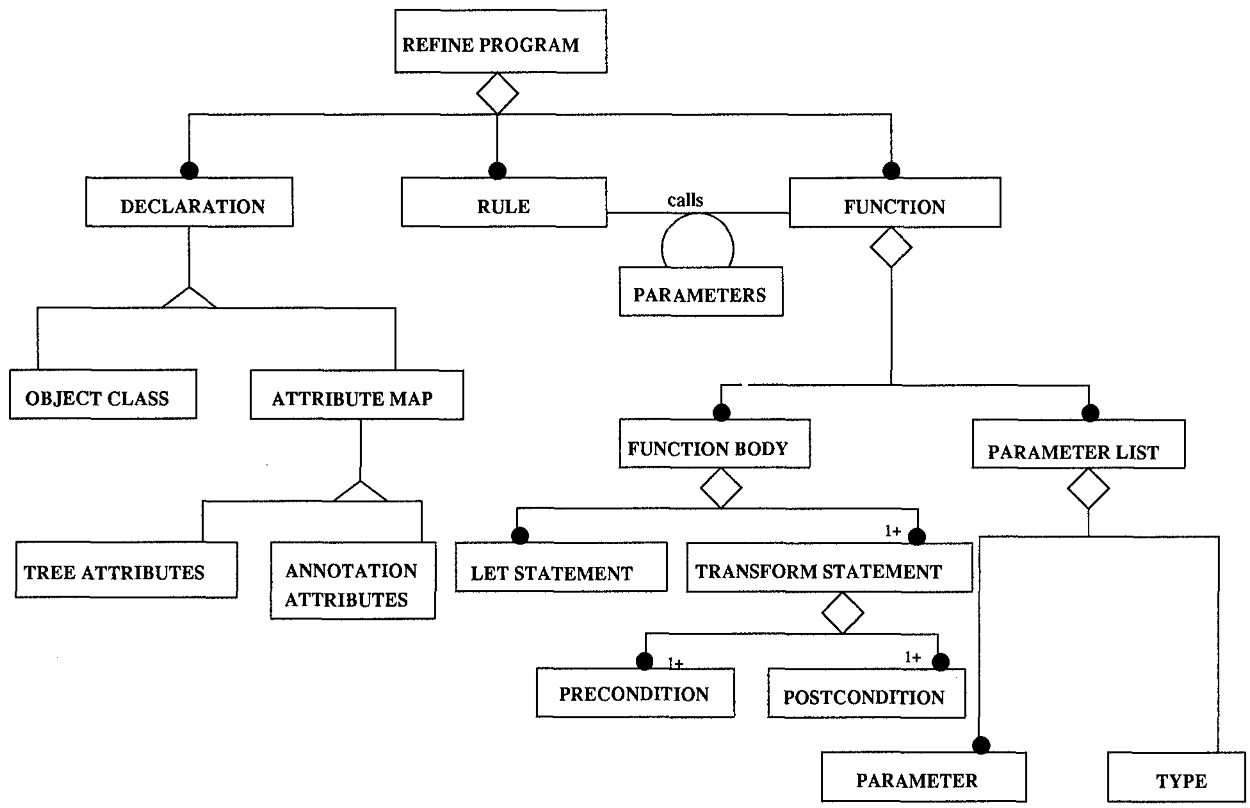

Figure K.1 Unified Domain Theory Model

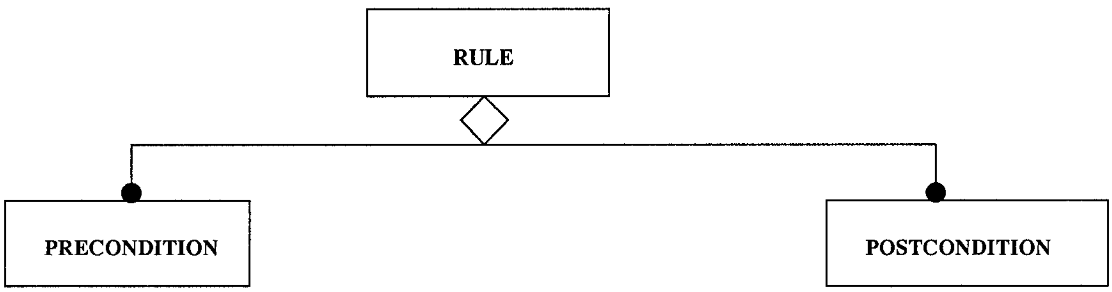

Figure K.2 Rule Object Class Model

Appendix L. REFINE *Code for the Target Domain Model*

The following REFINE specification defines the object classes and map attributes for the execution framework. Because the target execution framework is REFINE, the code incorporates the notions of object classes, map attributes, functions, and rules, as the target objects.

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: exec.re

Description:  The following program defines the transform rules for the formal
execution model.  Refine has been selected as the target execution environment.
Therefore, the transforms map the unified domain model into a refine execution
model.
||#

%------------------------------------------------------------------------------
% Refine Object Class Declaration
%------------------------------------------------------------------------------
var Refine-Object-Model                 : object-class subtype-of user-object
var Refine-Declaration                  : object-class subtype-of Refine-Object-Model

var Refine-Decl-Attrs                   : object-class subtype-of Refine-Object-Model
  var Refine-Annotation-Attr             : object-class subtype-of Refine-Decl-Attrs
  var Refine-Tree-Attr                   : object-class subtype-of Refine-Decl-Attrs
var Refine-Decl-Obj-Class               : object-class subtype-of Refine-Object-Model

var Refine-Function                     : object-class subtype-of Refine-Object-Model
var Refine-Function-Body                : object-class subtype-of Refine-Object-Model
var Function-Parameters                 : object-class subtype-of Refine-Object-Model
var Parameter-Value                     : object-class subtype-of Refine-Object-Model
var Parameter-Type                      : object-class subtype-of Refine-Object-Model
var Function-Let-Statement              : object-class subtype-of Refine-Object-Model
var Function-Transform                  : object-class subtype-of Refine-Object-Model
var Function-Precondition               : object-class subtype-of Refine-Object-Model
var Function-Postcondition              : object-class subtype-of Refine-Object-Model

var Refine-Rule                         : object-class subtype-of Refine-Object-Model
var Rule-Precondition                   : object-class subtype-of Refine-Object-Model
var Rule-Postcondition                  : object-class subtype-of Refine-Object-Model
```

```
%------------------------------------------------------------------------------
% Refine Attribute Declarations for Branches in Tree Structure
%------------------------------------------------------------------------------

var refineDecl           : map(Refine-Object-Model, Refine-Declaration)     = {||}
var refineObjClasses     : map(Refine-Object-Model, set(Refine-Decl-Obj-Class))
                               computed-using refineObjClasses(x)           = {}
var refineAttrs          : map(Refine-Object-Model, set(Refine-Decl-Attrs))
                               computed-using refineAttrs(x)                = {}

var refineFunctions      : map(Refine-Object-Model, set(Refine-Function))
                               computed-using refineFunctions(x)            = {}
var functionBody         : map(Refine-Object-Model, Refine-Function-Body)   = {||}
var parameterList        : map(Refine-Object-Model, set(Function-Parameters))
                               computed-using parameterList(x)              = {}
var parameters           : map(Function-Parameters, set(Parameter-Value))
                               computed-using parameters(x)                 = {}
var parameterType        : map(Function-Parameters, Parameter-Type)         = {||}
var letStatements        : map(Refine-Object-Model, set(Function-Let-Statement))
                               computed-using letStatements(x)              = {}
var transforms           : map(Refine-Object-Model, set(Function-Transform))
                               computed-using transforms(x)                 = {}
var functionPre          : map(Refine-Object-Model, set(Function-Precondition))
                               computed-using functionPre(x)                = {}
var functionPost         : map(Refine-Object-Model, set(Function-Postcondition))
                               computed-using functionPost(x)               = {}

var refineRules          : map(Refine-Object-Model, set(Refine-Rule))
                               computed-using refineRules(x)                = {}
var rulePre              : map(Refine-Object-Model, set(Rule-Precondition))
                               computed-using rulePre(x)                    = {}
var rulePost             : map(Refine-Object-Model, set(Rule-Postcondition))
                               computed-using rulePost(x)                   = {}

%------------------------------------------------------------------------------
%  Annotation Attributes For Object Refine-Decl-Attrs
%------------------------------------------------------------------------------

var attrDomain            : map(Refine-Decl-Attrs, symbol)                 = {||}
var attrCoDomain          : map(Refine-Decl-Attrs, symbol)                 = {||}

%------------------------------------------------------------------------------
%  Declarations for other Objects
%------------------------------------------------------------------------------

var Exec-Program          : Refine-Object-Model = make-object('Refine-Object-Model)
```

```
%---------------------------------------------------------------------------
%  Structure for Abstract Syntax Tree
%---------------------------------------------------------------------------

form Define-Tree-Attributes-of-Refine-Rules
  Define-Tree-Attributes('Refine-Object-Model, {'refineDecl, 'refineFunctions,
'refineRules})&
  Define-Tree-Attributes('Refine-Declaration, {'refineObjClasses,
       'refineAttrs})&
  Define-Tree-Attributes('Refine-Function, {'parameterList, 'functionBody})&
  Define-Tree-Attributes('Function-Parameters, {'parameters, 'parameterType})&
  Define-Tree-Attributes('Refine-Function-Body, {'letStatements,
 'transforms})&
  Define-Tree-Attributes('Function-Transform, {'functionPre, 'functionPost})&
  Define-Tree-Attributes('Refine-Rule, {'rulePre, 'rulePost})
```

Appendix M. REFINE *Code for the Initial Execution Code*

The following REFINE program creates an initial framework for an executable program. The main function, Make-Executable, encapsulates the Make-Object-Classes, Make-Attributes, and Make-Function operations generating the appropriate object classes and attributes in the execution framework from the source objects. After creating the execution framework, the Print-Program function traverses the executable AST and prints an executable program's initial shell consisting of the object classes, map attributes, function names, and parameters.

```
!! in-package("RU")
!! in-grammar('user)

#||
File name: u-transform.re

Description:  The following program creates the initial framework for the execution
program.  The print function prints the executable program consisting of object
classes, map attributes and function names.

||#

!! in-package("RU")
!! in-grammar('user)

function make-object-class (Temp-Specs         : set(DomainTheoryTypes),
    Temp-Decl         : Refine-Declaration,
    Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Refine-Object   : Refine-Decl-Obj-Class = make-object('Refine-Decl-Obj-Class))

  (enumerate tr over Temp-Specs do
   (enumerate o over descendants-of-class(tr, 'ObjectTheory) do
    Refine-Object <- set-attrs(make-object('Refine-Decl-Obj-Class),
       'Name, name(ot-id(o)));
    (format (true, "Refine-Object ~a~%", Refine-Object));
    refineObjClasses(Temp-Decl) <- refineObjClasses(Temp-Decl) with Refine-Object;
    (format (true, "programDecl ~a~%", Temp-Decl))
    )
   );
   refineDecl(Temp-Exec-Program) <- Temp-Decl;
   (format (true, "Exec-Program ~a~%", Temp-Exec-Program))
```

```
function make-attributes (Temp-Specs       : set(DomainTheoryTypes),
  Temp-Decl         : Refine-Declaration,
  Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Refine-Attr      : Refine-Decl-Attrs = make-object('Refine-Decl-Attrs))

  (enumerate tr over Temp-Specs do
   (enumerate o over descendants-of-class(tr, 'ObjectTheory) do
    (enumerate a over descendants-of-class(o, 'BasicDeclSeq) do
     (format (true, "In BasicdeclSeq ~a~%", a));
     (enumerate f over descendants-of-class(a, 'Identifier) do
      (format (true, "In Identifier ~a~%", f));
       Refine-Attr <- set-attrs(make-object('Refine-Decl-Attrs),
'Name, name(i-dent(f)),
'attrDomain, name(ot-id(tr)),
'attrCoDomain, any-expr(a));
       (format (true, "Refine-Attr ~a~%", Refine-Attr));
       refineAttrs(Temp-Decl) <- refineAttrs(Temp-Decl) with
       Refine-Attr;
       (format (true, "programDecl ~a~%", Temp-Decl))
      )
     )
    )
   );
   refineDecl(Temp-Exec-Program) <- Temp-Decl;
   (format (true, "Exec-Program ~a~%", Temp-Exec-Program))


function Make-Function (Temp-Specs       : set(DomainTheoryTypes),
Temp-Exec-Program : Refine-Object-Model) : Refine-Object-Model =

  let (Temp-Function   : Refine-Function = make-object('Refine-Function))
  let (Temp-Set        : set(Refine-Function) = {})

  (enumerate tr over Temp-Specs do
   (enumerate s over descendants-of-class(tr, 'StateTheory) do
    Temp-Set <- Temp-Set with (set-attrs(make-object('Refine-Function),
 'Name, name(st-id(s))))
    );
   (enumerate e over descendants-of-class(tr, 'EventTheory) do
    Temp-Set <- Temp-Set with (set-attrs(make-object('Refine-Function),
 'Name, name(et-id(e))))
    );
   (enumerate o over descendants-of-class(tr, 'FunctionalTheory) do
    Temp-Set <- Temp-Set with (set-attrs(make-object('Refine-Function),
 'Name, name(ft-id(o))))
    )
   );
   refineFunctions(Temp-Exec-Program) <- refineFunctions(Temp-Exec-Program) union Temp-Set;
   (format (true, "Exec-Program ~a~%", Temp-Exec-Program))
```

```
function Make-Executable (theory : DomainTheory) : Refine-Object-Model =

  let (Specs                 : set(DomainTheoryTypes) = {})
  let (Temp-Exec-Program     : Refine-Object-Model = make-object('Refine-Object-Model))
  let (Program-Decl          : Refine-Declaration = make-object('Refine-Declaration))

  (enumerate s over descendants-of-class(theory, 'DomainTheoryTypes) do
   Specs <- Specs with s
   );
  (Make-Object-Class(Specs, Program-Decl, Temp-Exec-Program));
  (Make-Attributes(Specs, Program-Decl, Temp-Exec-Program));
  (Make-Function(Specs, Temp-Exec-Program))

function Print-Program (program : object) =

  undefined?(program) --> format(true, "Nothing to print ~%");
  defined?(program) -->
    format (true, "!! in-package(RU) ~%");
    format (true, "!! in-grammar('user) ~% ~% ~%");
    (enumerate d over descendants-of-class(program, 'Refine-Declaration) do
     (enumerate o over descendants-of-class(d, 'Refine-Decl-Obj-Class) do
        format(true, "%---------------- ~A -----------------------% ~%", name(o));
        format(true, "%-----------------------------------------------% ~%");
        format(true, "var ~A : object-class subtype-of user-object ~%",
       name(o));
        format(true, " ~%")
      );
     (enumerate a over descendants-of-class(d, 'Refine-Decl-Attrs) do
        format(true, "var ~A   : map(~A, ~A) = {||} ~%", name(a), attrDomain(a),
       attrCoDomain(a));
        format(true, " ~%")
      )
     );
   (enumerate e over descendants-of-class(Program, 'Refine-Function) do
    format(true, "function ~A () = ~%", name(e))
    )
```

Appendix N. User's Manual for the UZed Parser

This appendix contains a listing of the Lisp files required to parse Z specifications. The order in which the files are listed below indicates the required compilation order. Following the file listings is a set of instructions guiding a user through a sample session for parsing Z specifications and viewing the corresponding ASTs. In addition, the final instruction shows the user how to create an execution framework for the parsed Z specifications.

```
1. Load system files for Dialect and Object Inspector.

  (load ''load-inspector'')
  (load-inspector)

2. Load the unified Zed domain model and grammar files and
   the target execution domain model.  Bring the unified Zed
   grammar in the environment.

  (load ''uzed-dm'')
  (load ''uzed-gram'')
  (load ''utoolkit-dm'')
  (load ''utoolkit-gram'')
  (in-grammar 'UToolKit)
  (load ''exec'')

3. Load the semantic analysis files.

  (load ''u-schema-inclusion'')
  (load ''u-delta-schemas'')
  (load ''u-delta-vars'')
  (load ''u-xi-schemas'')
  (load ''u-xi-vars'')

4. Load the translation program.
  (load ''u-transform'')

5. Parse your domain specification.
  % C-x-i u-counter.re (for counter domain)
  % C-x-i u-fueltank.re (for fuel tank domain)
  % C-x-i u-passenger-list.re (for aircraft passenger list domain)
```

```
% C-x-i u-datadict.re (for data dictionary domain)
```

6. View the initial domain specification using Object Inspector.

```
(pup)                                          % to view the current object;
                                               % returns the magic object number.
(insp::inspect(mcn object's magic-number))     % brings Object Inspector up
                                               % with the current object.
```

7. Execute the semantic analysis files on your domain specification to bring in all referenced schemas.

```
(make-schema-inclusion)
(make-delta-schemas)
(make-delta-vars)
(make-xi-schemas)
(make-xi-vars)
```

8. View the augmented AST using Object Inspector.

```
(insp::inspect-obj(mcn object's magic-number))
```

9. Execute the translation function.

```
(make-executable(mcn domaintheory object's magic number))
(print-program (mcn executable object's magic number))
```

10. Demo Script Sequence (C-x-i to bring domain specifications into Refine)

```
test counter domain   (~/thesis/compiler-code/uzed/valid-examples/u-counter.re)
test fueltank domain  (~/thesis/compiler-code/uzed/valid-examples/u-fueltank.re)
test passenger list   (~/thesis/compiler-code/uzed/valid-examples/u-passenger-list.re)
test data dictionary  (~/thesis/compiler-code/uzed/valid-examples/u-datadict.re)
```

The REFINE source code for Z and UZed may be obtained, upon request, from:

Maj Paul Bailor
AFIT/ENG
2950 P Street
Wright-Patterson AFB, OH 45433-7765

(513)255-9263
DSN 785-9263
email: pbailor@afit.af.mil

Bibliography

1. Aho, Alfred V., et al. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

2. Alencar, Antonio J. and Joseph A. Goguen. "OOZE." *Object Orientation in Z* edited by Susan Stepney, chapter 7, Springer-Verlag, 1992.

3. Bailor, Paul D. "Principles of Embedded Software." Class Notes: CSCE 693, Real-Time Embedded Systems, Fall 1994, Air Force Institute of Technology.

4. Bailor, Paul D. "Theories and Software Engineering." Class Notes: CSCE 793, Formal Methods in Software Engineering, Winter 1994, Air Force Institute of Technology.

5. Barden, Rosalind and Susan Stepney. "Support For Using Z." *Seventh Z User Meeting, London,1992*, edited by J.P. Bowen and J.E. Nicholls. Springer-Verlag, 1993.

6. Blaine, L., et al. SPECWARE^{TM} *User Manual*, 1994. For SPECWARE^{TM} Version Core4.

7. Bowen, Jonathan and Mike Gordon. "Z and HOL." *Eighth Z User Meeting, Cambridge, 1994*, edited by J.P. Bowen and J.A. Hall. 141 – 167. Springer-Verlag, 1994.

8. Breuer, Peter T. and Jonathan P. Bowen. "Towards Correct Executable Semantics for Z." *Eighth Z User Meeting, Cambridge, 1994.* 1994.

9. Brien, Stephen and John Nicholls. "*Z* Base Standard, Version 1.0." Distributed at the Seventh *Z* User Meeting, 1992.

10. Cohen, B. "A Rejustification of Formal Notations," *Software Engineering Journal*, 36–38 (January 1989).

11. de Meer, Jan, et al. "Introduction to Algebraic Specifications Based on the Language ACT ONE," *Computer Networks and ISDN Systems*, *23*(5):363–392 (1992).

12. Diller, Antoni. *Z, An Introduction to Formal Methods.* New York, New York: John Wiley and Sons, Inc., 1990.

13. Ehrig, H., et al. "Introduction to Algebraic Specification. Part 1: Formal Methods for Software Development," *The Computer Journal*, *35*(5):460–467 (1992).

14. Flynn, Mike, et al. "Formaliser - An Interactive Support Tool for Z." *Fourth Z User Meeting, Oxford, 1989*, edited by J.P. Bowen and J.E. Nicholls. Springer-Verlag, 1990.

15. Gaudel, Marie-Claude. "Algebraic Specifications." *Software Engineer's Reference Book* edited by John A. McDermid, chapter 22, CRC Press, Inc., 1993.

16. Guttag, John V. and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

17. Hartrum, Thomas C. and Paul D. Bailor. "Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies." *Software Engineering Education Proceedings.* Software Engineering Insitute (SEI), January 1994.

18. Jia, Xiaoping. *ZTC: A Type Checker for Z.* Institute for Software Engineering, DePaul University, Chicago, IL, 1994.

19. Lano, Kevin and Howard Haughton. "A Comparative Description of Object-Oriented Specification Languages." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Haughton, chapter 2, Prentice Hall, 1994.

20. Lano, Kevin and Howard Haughton. "Specifying a Concept-Recognition System in Z^{++}." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Haughton, chapter 7, Prentice Hall, 1994.

21. Lano, Kevin C. "Z^{++}." *Object Orientation in Z* edited by Susan Stepney, chapter 9, Springer-Verlag, 1992.

22. Lightfoot, David. *Formal Specification Using Z*. MacMillan Education Ltd, 1991.

23. Lin, Captain Catherine J. *Unification of* LARCH *and Z-Based Object Models To Support Algebraically-Based Design Refinement: The* LARCH *Perspective*. MS thesis, Graduate School of Engineering, Air Force Institute of Technology (AU), 1994.

24. Martin, James and James J. Odell. *Object-Oriented Analysis and Design*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1992.

25. McCain, Ron. "Reusable Software Component Construction: A Product-Oriented Paradigm." *AIAA/ACM/NASA/IEEE Computers in Aerospace V Conference*. 125–135. AIAA, October 1985.

26. McDermid, John, et al. "CADIZ - Computer Aided Design in Z." *Fifth Z User Meeting, Oxford, 1990*. 1991.

27. McMorran, Mike and Steve Powell. *Z Guide for Beginners*. Blackwell Scientific Publications, 1993.

28. Meira, Silvio Lemos and Ana Lucia C. Cavalcanti. "MooZ Case Studies." *Object Orientation in Z* edited by Susan Stepney, chapter 5, Springer-Verlag, 1992.

29. Meira, Silvio Lemos, et al. "The Unix Filing System: A Moo*Z* Specification." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Haughton, chapter 4, Prentice Hall, 1994.

30. Merriam-Webster, A. *Webster's Third World International Dictionary*. Merriam-Webster, Inc., 1981.

31. Monahan, Brian and Roger Shaw. "Specification and Mathematical Models." *Software Engineer's Reference Book* edited by John A. McDermid, chapter 21, Butterworth-Heinemann Ltd, 1991.

32. Norcliffe, Allan and Gil Slater. *Mathematics of Program Construction*. Ellis Horwood Limited, 1991.

33. Partsch, H. and R. Steinbruggen. "Program Transformation Systems," *ACM Computing Surveys*, *15*(3):199 – 235 (1983).

34. Partsch, Helmut A. *Specification and Transformation of Programs*. Springer-Verlag, 1990.

35. Ratcliff, Bryan. *Introducing Specification Using Z: A Practical Case Study Approach*. McGraw-Hill, 1994.

36. Reasoning Systems, Inc. DIALECT^{TM} *User's Guide*. Palo Alto, CA, July 1990. For DIALECT^{TM} Version 1.0.

37. Reasoning Systems, Inc. REFINE^{TM} *User's Guide*. 3260 Hillview Ave., Palo Alto, CA 94304, May 1990. For REFINE^{TM} Version 3.0.

38. Rose, Gordon. "Object-Z." *Object Orientation in Z* edited by Susan Stepney, chapter 6, Springer-Verlag, 1992.

39. Rose, Gordon and Roger Duke. "An Object-Z Specification of Mobile Phone System." *Object-Oriented Specification Case Studies* edited by Kevin Lano and Howard Haughton, chapter 5, Prentice Hall, 1994.

40. Rumbaugh, J., et al. *Object-Oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

41. Scharbach, P. N. *Formal Methods: Theory and Practice*. Boca Raton, Florida: CRC Press, Inc., 1989.

42. Schreiner, Axel T. and Jr. H. George Friedman. *Introduction to Compiler Construction with UNIX*. Prentice-Hall, 1985.

43. Semmens, Lesley and Pat Allen. "Using Yourdon and Z: An Approach to Formal Specification." *Fifth Z User Meeting, 1990*. 228 – 253. 1991.

44. Smith, Douglas. "KIDS: A Semi-Automatic Program Development System," *IEEE Software Transactions*, *16*(9):1024 – 1043 (September 1990).

45. Spivey, J.M. *The Z Notation*. London: Prentice Hall International, 1989.

46. Spivey, J.M. *The Z Notation, Second Edition*. Prentice Hall International, 1992.

47. Spivey, Mike. *A Guide to the Zed Style Option*, 1990.

48. Srinivas, Yellamraju V. *Algebraic Specifications: Syntax, Semantics, Structure*. Technical Report, University of California, Irvine, 1990.

49. Srinivas, Yellamraju V. "Algebraic Specification for Domains." *Domain Analysis and Software Systems Modeling* edited by Guillermo Arango and Ruben Prieto-Diaz, IEEE Computer Society Press, 1991.

50. Steggles, Pete and Jason Hulance. *Z Tools Survey*. Technical Report, Imperial Software Technology, 1994.

51. Stepney, Susan, et al. "Hall's Style." *Object Orientation in Z* edited by Susan Stepney, chapter 3, Springer-Verlag, 1992.

52. USAF. "Guidelines for Successful Acquisition and Management of Software Intensive Systems - Draft." AFPAM 63-115, 1993.

53. Valentine, Samuel H. *Enhancing the Z Mathematical Toolkit*. Technical Report, University of Brighton, England, 1993.

54. Valentine, Samuel H. "Putting Numbers Into The Mathematical Toolkit." *Seventh Z User Meeting, London,1992*, edited by J.P. Bowen and J.P. Nicholls. Springer-Verlag, 1993.

55. Valentine, Samuel H. *Highlights of the Z^{--} Mathematical Toolkit.* Technical Report, University of Brighton, England, 1994.

56. Valentine, Samuel H. *Operation Schemas in Z^{--}.* Technical Report, University of Brighton, England, 1994.

57. Valentine, Samuel H. *Translating Fortran Into Z^{--}.* Technical Report, University of Brighton, England, 1994.

58. Valentine, Samuel H. *The Z^{--} Language.* Technical Report, University of Brighton, England, 1994.

59. Whysall, Peter J. *Object Oriented Specification and Refinement.* PhD dissertation, University of York, York, England, 1991.

60. Whysall, Peter J. "Refinement." *Software Engineer's Reference Book* edited by John A. McDermid, chapter 24, Butterworth-Heinemann Ltd, 1991.

61. Whysall, Peter J. "Z Expression of Refinable Objects (ZERO)." *Object Orientation in Z* edited by Susan Stepney, chapter 4, Springer-Verlag, 1992.

62. Woodcock, Jim and Martin Loomes. *Software Engineering Mathematics.* London: Pitman Publishing, 1988.

Vita

Captain Kathleen May Wabiszewski was born on 12 May 1959 in Jersey City, New Jersey and graduated from Middletown North High School in Middletown, New Jersey in June, 1977. She enlisted in the Air Force in December 1980 and completed technical training for ground radio maintenance at Keesler AFB, Mississippi in September 1981. She spent eighteen months as a radio maintenance technician at Fort Fisher AFS, North Carolina and then became an Engineering Team Chief at the 1839 Engineering and Installation Group at Keesler AFB, Mississippi. Remaining at Keesler AFB, she was a certified Master Instructor of ground radio maintenance from November, 1983 until September, 1989. Topics of instruction included electronic theory, logic, circuit operation, and component troubleshooting of UHF/VHF receivers, transmitters, and transceivers. During her tour as an instructor, she attended college at night and, in May, 1988, she graduated, with honors, from the University of Southern Mississippi with a Bachelor of Science degree in Mathematics. In September, 1989, she entered Officer Training School and received her commission on 22 December 1989. As a new lieutenant, she was assigned to the 1912 Computer Systems Group, Langley AFB, Virginia, in May, 1990, as a Test Manager for Joint Tactical Systems. She was responsible for all facets of interoperability testing of Air Force drug-interdiction systems. From April, 1992 until April, 1993, she served as the Executive Officer to the 1912th Group Commander. In May, 1993, First Lieutenant Wabiszewski entered the Air Force Institute of Technology at Wright-Patterson AFB, Ohio, to pursue a Master of Science degree in Computer Systems. While attending AFIT, she was awarded a Master of Science degree in Systems Management from Florida Institute of Technology, where she attended classes during her assignment to Langley AFB. Upon graduation, Captain Wabiszewski will be assigned to the AFC4 Agency, Scott AFB, Illinois, as a member of the Software Capability Assessment Team.

Permanent address: 215 Woodridge Court
Collinsville, IL 62234

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1994

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
UNIFICATION OF LARCH AND Z-BASED OBJECT MODELS TO SUPPORT ALGEBRAICALLY-BASED DESIGN REFINEMENT: THE Z PERSPECTIVE

5. FUNDING NUMBERS

6. AUTHOR(S)
Kathleen May Wabiszewski, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/94D-24

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Mr. Glenn Durbin
NSA/Y23
9800 Savage Road
Fort Meade, MD 20755-6000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research established a foundation for formalizing the evolution of *Z*-based object models to theories, part of a dual approach for formally extending object-oriented analysis models using the *Z* and LARCH languages. For the initial phase, a comprehensive, consistent, and correct *Z* language parser was implemented within the SOFTWARE REFINERY^{TM} Programming Environment. The *Z* parser produced abstract syntax trees (ASTs) of objects, thereby forming the basis for analyzing the similarities and differences between the *Z*-based and LARCH-based object representations. The second phase used the analysis of the two languages to identify fundamental core constructs that consisted of similar syntactic and semantic notions of signatures and axioms for describing a problem domain, thereby forming a canonical framework for formal object representations. This canonical framework provides a front-end for producing design refinement artifacts such as synthesis diagrams, theorem proving sentences, and interface languages. The final phase of the process demonstrated the feasibility of interface language generation by establishing an executable framework that mapped *Z* into the SOFTWARE REFINERY^{TM} Environment to rapidly prototype object-oriented *Z* specifications.

14. SUBJECT TERMS
software engineering, specifications, formal specification languages, application composition systems, Z specification language, Z language parser

15. NUMBER OF PAGES
213

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL